Cryptographic Access Control in a Cloud Based File Storage Environment

Mads Lundt & Kristian Flamsted

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

The goal of this master thesis is to look at the possibility to develop a cloud based file storage system. This system should include a local client and a remote file storage system, using MediaWiki.
The purpose is to explore whether it is a possibility to use cryptographic access control, to make sure only people with the correct cryptographic keys, are getting the right access. Avoiding the use of any regular access control methods that already exist. When using cryptographic access control the all data is encrypted already at the client and this allows the trust level to the server to be much lower than in any regular cloud file storage setups. This is because nobody should ever be able to read the data at any time if they do not have the correct keys. When protecting the data using cryptographic access control a combination of symmetric- and asymmetric cryptography is used.

The possibility of sharing files among different users is also a part of this project, and this can be done using key rings. Each user have access to key ring, that includes keys for both files and for other key rings and these can be shared. The system is supposed to automatically create these keys when creating files, but sharing is something a user does manually.

The current prototype implemented shows that it is possible to create a cloud based file storage using MediaWiki. However, the current application is not using FUSE to automatically create keys when adding files due to errors. The reason is that the XML parser was not working properly with FUSE. However, as it was close to a solution so the FUSE library have been saved for possible future implementations. It is possible to share files in between users, but this needs to be done manually.
A risk analysis has been made showing possible attacks, how they are done and

how big of a risk these are.

Resume

Målet for denne afhandling er at undersøge mulighederne for at udvikle et cloud baseret fil system. Hensigten er at bruge FUSE, som den lokale klient for at skabe et fil system og herefter bruge MediaWiki som den eksterne server, hvor filerne gemmes.
Formålet med projektet er at se om det er muligt kun at bruge kryptografisk adgangskontrol for at sørge for det kun er folk med de korrekte kryptografiske nøgler der for de rette adgange. Den overordnet idé er helt at undgå allerede eksisterende adgangs kontroller. Når der bruges kryptografisk adgangskontrol kan det sikres at alt data bliver krypteret på den lokale klient, hvilket betyder at man ikke behøver at stole på serveren, som ved normale cloud baseret fil systemer. Dette er fordi at uanset hvor dataen ender, vil det kun være dem med de rette nøgler der kan læse dataen. I kryptografisk adgangskontrol bruges der en kombination af symmetrisk- og asymmetrisk kryptografik for data beskyttelse.

Det skal også være muligt at dele filer i mellem flere brugere og til dette formål bruges der nøgle ringe. Hver bruger har deres egen nøglering, indeholdende en andre nøgler, der både kan være nøgler til filer eller links til andre nøgleringe. Idéen er at systemet automatisk skal oprette nøgler for hver fil, som brugeren laver dem. Disse kan deles mellem brugererne, men dette skal gøres manuelt.

Den nuværende implementeret prototype, viser at det muligt at lave et cloud baseret fil system ved brugern af MediaWiki. Der bliver dog ikke gjort brug af FUSE til automatisk at oprette nøgler, da dette biblotek har givet fejl. Grunden til dette var at XML parseren ikke fungerede sammen med FUSE. Da det har været tæt på at FUSE problemerne kunne løses, er kildekoden til dette blevet gemt, som en del af projektet. Det er muligt i denne implementering at dele filer mellem brugere, men dette kræver en række manuelle operationer.

Derudover er der lavet en risiko analyse hvor mulige trusler til systemet er gennemgået og hvor stor en trussel disse er til systemet.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with cloud based file storage and was done in the period from January 4th, 2016 to June 4th, 2016 and was supervised by associate professor Christian D. Jensen.

Lyngby, 04-June-2016

[illegible] [illegible]

Mads Lundt & Kristian Flamsted

Acknowledgements

We would like to thank our supervisor Christian D. Jensen for help, advise and for giving great inputs and ideas during this project.

Contents

Chapter 1

Introduction

The cloud is everywhere and more applications are turning into cloud based applications. The reason for this is that people need their pictures, videos and so on, available at any time and everywhere. Cloud based file storage solutions is a thing that has arrived to stay, but the data needs to be protected, while sending, receiving and storing it as it should not be available to any other than the owner(s). However, the owners and administrators of the data should be able to share it with whomever they want to. Meaning it should be available to any person they desire.
To solve the problem of cloud based file storage it is suggested to use cryptographic access control combined with proper key management, to make sure that only the correct users can get access to the data. This involves encryption of the data before it is transmitted to the remote server. This means that the data stored on the remote server is always encrypted.
In this project the cloud based file storage problem is solved using MediaWiki as the remote server storage together with proper key management. This gives an environment that is easy to setup for everyone as well as a very clean cryptography solution. All data handled is encrypted already at the client which means that authentication of the server is not necessary.
From this project it has been learned that a cloud based file storage is a really complex problem involving many sub problems. Even when creating a simple clean solution it gets complex with a lot of needed choices to be made.

1.1 Problems

The goal of this project is to create a very easy to setup cloud based file storage solution. Combining cryptographic access control together with proper key management giving a very clean cryptography setup. MediaWiki should be used for the remote storage and FUSE for the local file system. The goals/problems of the project can be described as follows:

- Implement a cloud based file storage using MediaWiki backend.
- Uploading and downloading data to and from the MediaWiki
- Protect the data, so it is only available to the owner(s).
- Implement a way for a user to share files with another user, without a third part getting access to the same data.
- Implement a client file system using FUSE.

1.2 Report Structure

The rest of this report includes the following chapters.

Theory The theory chapter is where all the basic knowledge needed for this project is outlined. It is build up by eight different sections. The general cryptography section including symmetric- and asymmetric cryptography as well as hashing. The access control session researching different access control methods. The next section, key rings, gives the idea of how the key rings works, followed by how to share keys. MediaWiki is looked into next followed by existing encrypted file systems as well as FUSE.

Requirements This chapter is to define all the different requirements needed for the application. This is both functionality- and security requirements.

Design This chapter is used to design the prototype of the application, which then afterwards can be implemented. The cryptography choices needed is also done here.

Implementation This chapter shows how the already outlined application design has been implemented.

Results The results chapter is where the different tests needed are outlined and how they went. Furthermore, the security evaluation of the application is done here and afterwards the future work that could be done on this project is discussed here as well.

Conclusion The conclusion chapter is where the whole project is concluded and explains why this projects is a solution to the outlined problems.

Appendix The appendices includes a usage guide and performance tests.

Chapter 2

Theory

When implementing the Cloud File storage, a lot of basic information is required. The purpose of the theory chapter is to get all this information, to easier setup the requirements later. Especially general cryptography subjects such as symmetric, asymmetric and hashing are needed to ensure the goals of the project. However, cryptography is not the only things essential for this. The needed tools such as FUSE, MediaWiki and more will be covered as well.

2.1 General Cryptography

Cryptography is an important aspect in this project, because it only allows people with the secret, to the data, to read or write to it. Cryptography is using mathematics to encrypt and decrypt data. When encrypting it turns plain text into unrecognizable gibberish. This outcome is called cipher text. When reading the encrypted data there is a need to reverse the process of encryption and this is called decryption and turns the cipher text back into plain text.

The main focus for this project is to make sure only the owner(s) of the file is able to read and write to the file. For this to happen a deeper analysis of different cryptography methods have to be made.

2.1.1 Symmetric Cryptography

In symmetric cryptography the keys used for encryption and decryption are the same. It is possible for the keys to be identical or to just have a simple transformation between them. This means that the encryption and decryption algorithms are basically identical, except that the decryption process is done in the reverse order than the encryption.
Symmetric cryptography can use either stream ciphers or block ciphers.

2.1.1.1 Stream Ciphers

A stream cipher combines plain text with a pseudo random cipher digit stream (key stream). This means that each digit in the plain text is encrypted one at a time with the corresponding digit in the key stream.

- Stream ciphers are typically faster.
- They typically require less memory because it only works on one digit at a time.
- Not as susceptible to noise in transmission because each digit is encrypted individually.
- Does not need padding.

2.1.1.2 Block Ciphers

A block cipher is operating on fixed-length groups (blocks). This means that it takes a block of plain text and encrypts it to a block of cipher text. It is important to mention that it only operates on complete blocks. If a block is not complete some generated padding is added to the block to complete it. Some advantages over stream ciphers are:

- Easier to implement correctly.
- Depending on mode, it can provide integrity protection in addition to confidentiality.

This does not mean that stream ciphers are better than block ciphers or the other way around. Choosing between stream and block ciphers depends on the problem. A stream cipher would for example usually be better when the data is unknown or continuous (network stream) and a block cipher would usually be better where the amount of data is known from the beginning.

2.1.1.3 AES

Advanced Encryption Standard (AES) [NISb] has been established as one of the most used standards when doing symmetric encryption / decryption. The algorithm is used to make data impossible to use for people that do not have the key. AES have a fixed block size of 128 bits, and the key sizes possible are: 128, 192 and 256.
The cipher works with four simple methods: SubBytes, ShiftRows, MixColumns, and AddRoundKey.
These will only be briefly described and a more technical description can be found in the NIST paper (see [NISb]) section 5.

SubBytes
The SubBytes method is a substitution of bytes, where all bytes are transformed using a substitution table, also called S-box, that should be possible to invert. The inverted S-box is used for decryption. The byte are substituted by looking up in the S-box to see what the byte should be substituted with.

ShiftRows
The ShiftRow operation runs trough the block and simply shift all the rows depending on which row the operation is currently at. The first row is not shifted, the next row is shifted by one, third row by two and so on and all shifting of bytes is to the left in the byte array.

MixColumns
This transformation is working through each column of the blocks and then the block is worked as a four-term polynomial where each polynomial is multiplied with the polynomial

$$a(x) = 03x^3 + 01x^2 + 01x + 02 \mod x^4 + 1$$

This is multiplied using matrix multiplication, as the bytes are represented in a matrix.

AddRoundkey
The AddRoundKey methods are simply adding a key to the current state of the

cipher by XORing it. The key added is called a round key as it is different each round.

AES Algorithm
Initially AES starts everything with one AddRoundKey method.
The next thing is to run the four methods in the given order: SubBytes, ShiftRows, MixColumns and AddRoundKey 10, 12 or 14 times depending on the key size.
After this a last round consisting only of the transformations: SubBytes, ShiftRows and AddRoundKey is done.
The number of times running the rounds are 10 if the key size is 128, 12 if the key size is 192 and 14 times if the key size is 256.

A key of a bigger size is more complex and requires more time to crack. By looking at it mathematically a key of size 128 have 2^{128} combinations compared to a key size of 256 with 2^{256} combinations. With all these combinations it shows that brute forcing is just a waste of time and even if a super computer or a bot net could try ten billion keys a second on an AES 256, it would take almost 3 billion years to run through all key combinations.

2.1.1.4 Modes of Operation

When having an algorithm such as AES, a mode of operation is needed. Modes of operation are algorithms that uses block ciphers to be able to ensure things as authentication. Most of the modes of operation uses an initialization vectoralso called IV, which is a unique string of binary representations. As the sequence is unique, the IV, cannot be used twice.
There exists a lot of different modes of operation but the most common ones that will be described here are:

- Electronic Codebook (ECB)
- Cipher Block Chaining (CBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter (CTR)

Electronic Codebook (ECB)
Perhaps the simplest mode of operation is the ECB where the plain text message is split into blocks and then each block are encrypted separately. For decryption it is the same but vice versa.
The problem with ECB is that if two blocks in the plain text are the same they

will be encrypted into the same cipher text which means that it does not hide patterns in messages. The classic example for showing the bad side of ECB is when this is done for bitmap images, looking at the encrypted image, one can recognize the original image.
It is not recommended at all to be used in cryptography protocols as it is.

Cipher Block Chaining (CBC)
A very common and used mode of operation is the Cipher Block Chaining. In CBC the plain text blocks will be XORed with the cipher text output from the previous block. The first block of plain text, will however be XORed with an IV, which has to be unique.
The decrypting is very similar to the encryption just vice versa. When decrypting with a wrong IV it is not only the first block of the outputted plain text that will be wrong, but all of it because the blocks depend on each other. The reason for this is that the first block output is needed for the second block decryption, second block output is needed for the third block and so on.
Even though CBC is the most commonly used mode of operation, it still has one performance problem. It is not possible to parallelize CBC when encrypting which can be very effective sometimes as encryption can be very resource needed.

Cipher Feedback (CFB)
Looking at Cipher Text Feedback it is similar to the CBC. However, when using CFB the idea is to start with the IV and then encrypt this with the key then XOR that with the plain text to get the cipher text. The cipher text is then used as the IV for the next block and so on. The decryption is same as the encryption but vice versa. It is only the decryption that can be parallelized and this can be a problem sometimes.

Output Feedback (OFB)
In the Output Feedback mode it uses freshly generated blocks for each round, instead of using the cipher text blocks as in CBC. Starting with an IV through the block cipher encryption, and this is used as the next input. Post encryption in each round is XORed with the plain text creating the cipher text.
The decryption is basically the same as encryption with the cipher text and plain text being switched. A problem about OFB is that neither encryption or decryption is parallelizable as all outputs from one block is needed to the next. However, the block cipher encryption steps can be performed beforehand which can be beneficial.

Counter (CTR)
The Counter mode is the last mode of operation that will be looked at here and it is the only one that can be parallelized for both encryption and decryption.

The way CTR works is that there is a nonce created, which can somewhat be compared to an IV. This is then concatenated with a counter variable and becomes the block for the encryption of this key, this is XORed with the plain text to give the cipher text.
The decryption is basically the same as the encryption switching the cipher text and plain text.
The requirement of the counter variable is that it can never be the same for any of the blocks. This means the simple starting from 0 and then adding one for each block is often used.

2.1.2 Asymmetric Cryptography

In asymmetric cryptography the keys used for encryption and decryption are separated into two different paired keys called public- and private-key. The public-key is disseminated widely which means it is allowed to be publicly available. The private-key is only known to the intended recipient.
Comparing asymmetric with symmetric cryptography shows that there are some disadvantages and advantages:

- Asymmetric encryption is much, much slower than symmetric ciphers.
- There is no shared secret with asymmetric cryptography. This means you do not have to trust all involved parties to keep the key secret.

Depending on the problem asymmetric-keys can be better than symmetric-keys and the other way around.

2.1.2.1 RSA

RSA[RA83] is an asymmetric encryption algorithm that is used to encrypt and decrypt messages. The security of the algorithm is based on the fact that finding factors of a given integer is a hard problem. In RSA the product of two large prime numbers are published together with an extra value to be the public-key. However, the factors of these two prime numbers are kept secret.

Calculating Keys

1. Choosing two different random prime numbers, called p & q.

2. Calculate $n = pq$.
3. Calculate the totient $\phi(n) = (p-1)(q-1)$.
4. The public-key e exponent is now chosen such that $1 < e < \phi(n)$ and is a co prime with $\phi(n)$.
5. The secret key exponent d should be computed such that it satisfies: $de = k\phi(n)$, for an integer k.

The public-key can now be created by the modulus of n and e, and the private-key can be created by the modulus of n and d, which is stored as a secret.

Encryption
The encryption is done by having the message m and the public-key consisting of n & e, and the cipher text can be calculated with the formula:

$$c = m^e \mod n$$

Decryption
When calculating message m the private-key d is needed, and m can be calculated as follows:

$$m = c^d \mod n$$

However, RSA can give some troubles when not using an agreed upon padding scheme, but this will not be covered in the project.

2.1.3 Hashing

Hashing functions are very important in terms of cryptography and in general they are much used in a lot of different communication protocols. A hash function is a function that when given any data as input, it will output the data to a fixed size. The output is called a variety of things like; hash- value, code or sum. Given the same input, the output of the hash function should always be the same.
Looking into a few cryptography hash functions is done and the most popular ones today:

- MD5
- SHA-1
- SHA-2

The difference between a regular hash function and cryptography hash functions, is that the cryptography ones are practically impossible to recreate the input data (to the hash function) from the output hash. This give various of possibilities in what they can be used for; password verification, file manipulation detection, partly of setting up communication protocols and much more.

2.1.3.1 MD5

MD5 is a message-digest algorithm and it is commonly mentioned throughout a lot of cryptography papers. It is generating a 128-bit hash value from any given value, typically a string. It was published in 1992 and designed by Ron Rivest but already in 1996 a flaw was found but was not deemed as a fatal weakness. In 2004 it was shown that MD5 was not collision resistant and later it was considered as broken. This means that the same hash could be calculated from different given inputs bigger than 128 bit.

2.1.3.2 SHA-1

SHA-1 was published in 1995 and designed by United States National Security Agency (NSA). It is today widely used, but already in 2005 crypt analysts found theoretical attacks and said that it might not be secure enough and that a stronger hash function would be desirable. Microsoft, Google and Mozilla have announced that they will stop supporting SHA-1-based SSL certificates because of the insecurities in SHA-1.

2.1.3.3 SHA-2

SHA-2 is not one but a set of hash functions developed by the NSA. The SHA-2 contains six different hash functions; SHA- 224, 256, 384, 512, 512/224, 512/256. The six different algorithms have different length of inputs and outputs, hence the name indication. It is in the same series as SHA-1 and even though it has some similarities with SHA-1, the security flaws of SHA-1 have not been found in SHA-2. However, there exist various attacks on the SHA-2 family of algorithms, however these are usually very complex or only on small rounds versions of the algorithm, meaning the algorithm is considered secure for now.
The latest hash function in the Secure Hash Algorithm family is SHA-3, that was released by NIST on August 5, 2015. SHA-3 is not created to replace SHA-2, as there has not been demonstrated any significant attacks on SHA-2.

2.2 Access Control

The access control section is looking at a variety of different access control possibilities. Starting with cryptographic access control, as it can be somewhat considered an introduction to access controls in general, followed by more concrete examples as ACL, RBAC, ABAC and DAC.

2.2.1 Cryptographic Access Control

Access control is an important aspect in this project because it controls the permissions assigned to each user. If a user has access to a file, the user can have the following capabilities:

- Read and write
- Read

This could be based on asymmetric cryptography, where files are encrypted with a private-key (encryption key) and decrypted with a public-key (decryption key). This makes sure that only the person with the encryption key, for the file, can write to the file. People with the decryption key can read the file. This can be used to act as capabilities for the files by having the decryption key corresponds to a read-capability and the encryption key corresponds to a write-capability.[HJ03]
The cons for this approach is that asymmetric cryptography is not sufficiently fast compared to symmetric cryptography, because of the encryption and decryption algorithms.
The size also matters for the cryptogram made by the encryption. An asymmetric cryptography encryption increases the size of the cryptogram where a symmetric cryptography encryption does not. The asymmetric cryptography was only designed for encrypting data smaller than its key size.

A second approach is to use symmetric cryptography to encrypt and decrypt files [HJ03]. The problem by only using symmetric cryptography is to have capabilities for the files, such as read- and write-capability. The symmetric cryptography only allows a client with the belonging symmetric-key for the file to encrypt and decrypt it.
To solve this problem the same asymmetric cryptography needs to be extended to this approach but only on a smaller part. This allows the client to create and encrypt a digital signature (hash or message digest of the encrypted file) belonging to a file to guarantee integrity of the data. By each user having a

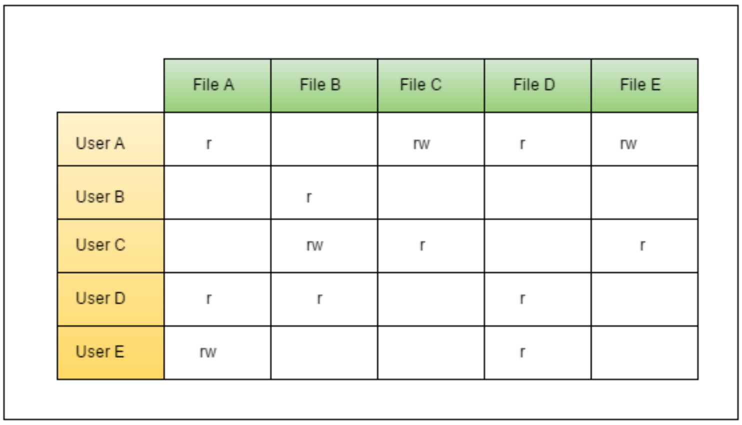
digital signature it is possible to use it to generate a signature for each file. The client and the server can identify if the file has been tampered with, by matching the file's signature. This give three keys:

- The **symmetric-key** used to encrypt and decrypt a file client-side. Only known by the client.
- The **encryption-key** used to encrypt the generated signature for the file, to ensure integrity of the file client-side. Only known by the client.
- The **decryption-key** used to verify the signature for the file, to prove the integrity of the file. This can be done by either client, server or both.

It is important to state that all files on the server and transmission between the server and client are encrypted. This means that the client needs to encrypt all data transmitted to the server. The idea is that the server does not do any decryption at all, however some server side verification could be implemented if wanted.

2.2.2 Access Control List

Access Control List (ACL) is an old and much used way of determine access to different objects. The idea is that the object (a file for example) has an ACL attached to it. The ACL can then determine all the users with read and/or write access, which are the operations that the user is allowed to have on that object.
This can be defined as a big table, with the users as rows and each file, object or operation as objects, and then for each of the objects, the permissions can be set into the table at that specific point. An example of this can be seen in figure 2.1.



	File A	File B	File C	File D	File E
User A	r		rw	r	rw
User B		r			
User C		rw	r		r
User D	r	r		r	
User E	rw			r	

Figure 2.1: Example of an access control table.

In many cases ACL is combined together with roles giving a role-based access control.

2.2.3 RBAC

Role-based access control (RBAC) is used to restrict system access to authorized users depending on what role they have. By having roles and assigning each role some permissions, it is possible to assign each user a role. This does not assign users to permissions directly but through their role. This simplifies the management of individual user rights but limits the flexibility because it does not scale well with many users. The limited flexibility is about having roles that contain all the different user rights. This is not a problem when having a few number of roles, but as the number of connected systems grows, the number of roles grows as well. This requires a huge backend implementation and can easily end up with having too many roles. This just transforms the problem from having many users to have many roles which are not really helping. This is also called role explosion [EK].
RBAC gets tricky when having more working sites. These often require the same roles on each site, but with a little difference and this means copying all of the roles when creating a new site.
When hiring new people for the same job, there might be needed something extra for the new person and that result in a new role as well, even though the new role could have been avoided.

2.2.4 ABAC

The before mentioned RBAC (see section 2.2.3) can in some way be considered an Attribute Based Access Control (ABAC)[VCHS], as the role for each user could be the attribute or a role could include different attributes. The way ABAC differs from RBAC, is that everyone have their own complete set of rules or attributes which define their different access. Obviously this leads to redundancy as more users need to have the exact same set of attributes.

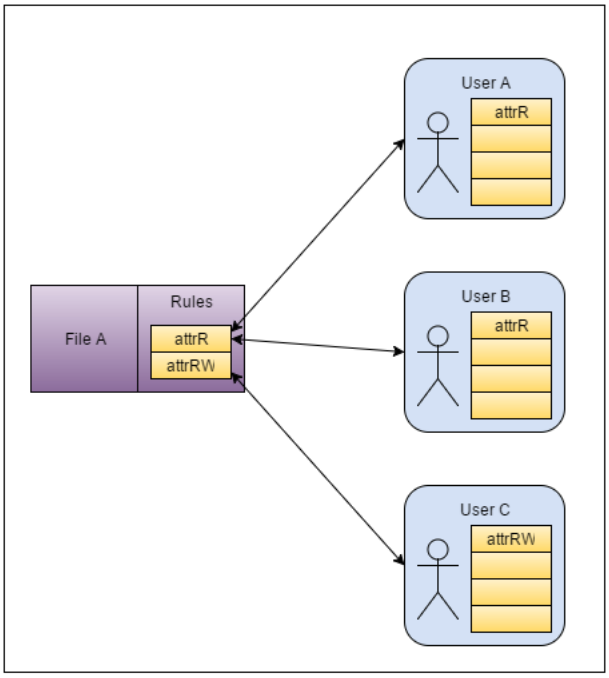


Figure 2.2: Example of attribute based access control

The ABAC is more flexible than RBAC, as when changing one persons access to the system (in RBAC), if the change is not an already existing role, a whole new role needs to be created. However, in ABAC the attribute will just be set to true or added to the user's attributes (depending on the setup). This will also avoid role explosion mentioned in the RBAC section 2.2.3.
Whenever a new method or process needs to be added, and it should be decided whether or not who should have access, a new attribute can be created and by default no one or a very few could initially have access, then as the users need the access it could be provided along the way. It does not need to be decided for all users right away who needs access.

The flexibility is way better than in for example RBAC, but ABAC is rather new compared to other Access Control Systems and there for it can be difficult to find good implementations when consider using it.

2.2.5 DAC

Discretionary access control[Jor] also called DAC is defined by TCSEC[1] by restricting access to objects based on user identification with supplied credentials during authentication. It is discretionary because the owner determines object access privileges. This means the owner can transfer information access to other users. With this approach each data object has an owner who determines the access policy for the data object.
A huge disadvantage by using DAC is that it is vulnerable to inherent vulnerabilities, such as a trojan horse.
An example of this can be done with two users A and B, where A is an honest user and B is a dishonest user.
A has a data file called **adatafile** with highly sensitive data. A makes sure that he is the only user to read the file which means no other users are authorized to access the file.
B is curious and determines to gain access to A's file, **adatafile**. Since B has legitimate access to the system, it allows him to implement an utility program. In this utility program B embeds a covert function (trojan horse) to read **adatafile** and copies its content into a another file in B's address space called **bdatafile**. **bdatafile** has an ACL associated with it that allow processes executing on A's behalf to write to the file and allowing B's process to read the file.
By convincing A to execute B's utility program, without knowing about the covert function, A does not know that his **adatafile** is being copied to **bdatafile**. The problem here is that the copy taking place is within the constraints of the DAC, and A is not aware of what is happening. This allows B to read all content that A writes to his own file **adatafile**.

[1]Trusted Computer System Evaluation Criteria is a United States Government Department of Defense (DoD) standard that sets basic requirements for assessing the effectiveness of computer security controls built into a computer system. The TCSEC is used to evaluate, classify and select computer systems being considered for the processing, storage and retrieval of sensitive or classified information.

2.3 Key Rings

Key rings are a way to store known encryption- and decryption-keys. It can be used in a cloud file storage environment to give a user access to a file if the user has a belonging key to the file. Each key in the key ring can point to either a file or another key ring.
Each key contains an encryption- and a decryption-key and the capabilities for the file. These capabilities could be read and/or write permissions. Then by combining all the features in key rings, it could end up giving an access control system.

2.3.1 Key Ring idea

The idea in this project is to use key rings to indicate whether or not a user have access for a file.

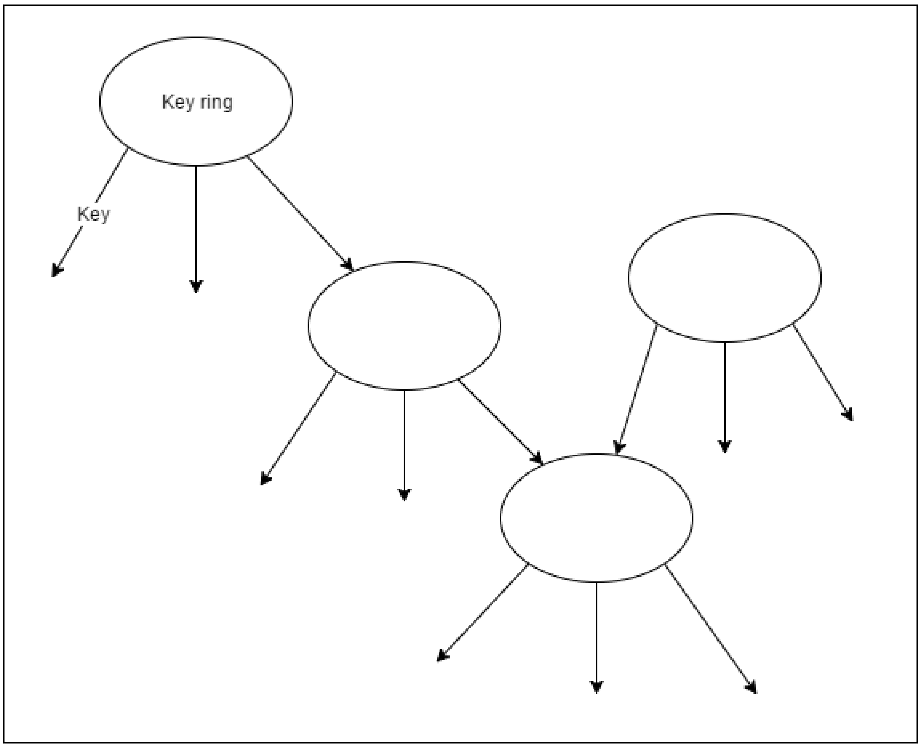


Figure 2.3: The key ring idea showed in a simple sketch.

Looking at the figure 2.3; the rings symbolize a key ring and the arrows are keys. A user can have several keys and key rings, and a key ring can lead to another key ring.
Having this kind of structure would end up being somewhat similar to a mix of role-based access control (RBAC) and attribute-based access control (ABAC). Each key ring could be representing a role in RBAC and a key could be an attribute in ABAC.

2.4 Key Sharing

Sharing keys among users require cryptography privacy and authentication for the data communication. This is required to make sure the key shared between two parties is only known by them.
A solution to do this is to use PGP (Pretty Good Privacy). PGP is a hybrid crypto system, which is by combining the convenience of asymmetric cryptography with the efficiency of symmetric cryptography. PGP is often used when signing, encrypting and decrypting text strings, emails, files, etc.

2.4.1 How PGP works

PGP is using both asymmetric- as well as symmetric cryptography to encrypt and decrypt the data. PGP starts by having the receiver generating two keys using asymmetric cryptography including a public- and private-key. The public-key is given to the sender where it is assumed that the connection for this transmission is secure and that the public-key has not been tampered with or read by any third-parties.
As seen in figure 2.4 this gives a decryption process which initializes the sharing by the receiver and an encryption process used by the sender.

The sender is now able to generate a random secret and encrypt the data, that is later sent to the receiver, by using symmetric cryptography. After the encryption of the data, the sender also encrypts the random generated secret with the asymmetric-key received by the receiver earlier. This means the sender now have an encrypted message containing the encrypted data and encrypted secret. The sender can send this to the receiver and the receiver can decrypt the secret with the private-key belonging to the public-key that was used for the encryption of the secret. With the secret decrypted the receiver is now able to decrypt the data.

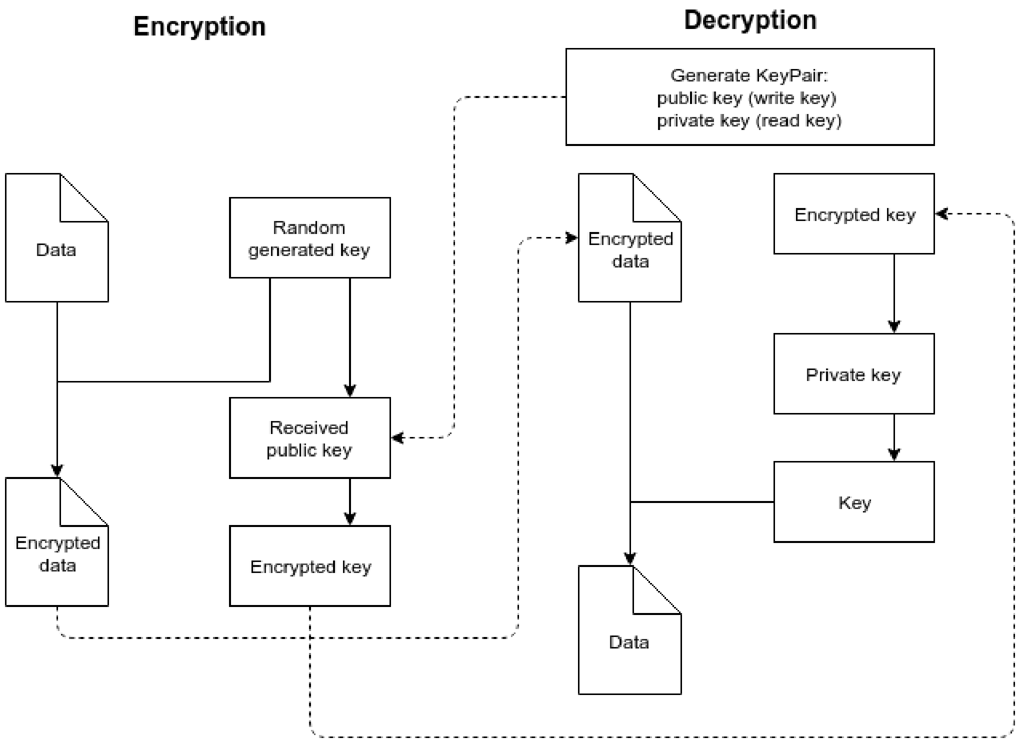


Figure 2.4: PGP

If a third-party should read the encrypted data object sent from the sender, the third-party is not able to decrypt or modify it. If the third-party modifies it, the receiver will know that this was not encrypted with the public-key provided in the beginning. This is only true because of the assumption that the public-key sent in the beginning was not read or tampered with by any third-parties.

2.5 MediaWiki

In the introduction and goals section 1 it was mentioned that MediaWiki is used as the backend for the file storage. In this chapter the MediaWiki will be described including; the API and how the design is and briefly how the revisions work. It will also be described how all this is supposed to be used as a part of the project.

2.5.1 General Description MediaWiki

MediaWiki is a free open source software developed by the Wikimedia foundation[wik] and it is the same engine that is used for their project Wikipedia[Fou], that have a lot of visitors everyday from all around the world. It is written in PHP[php] and is using MySQL[mys] as the database solution. A good thing about MediaWiki is that it is easy to setup, and is cross platform supported, as it can be setup on Windows, MAC OS and Linux as well. The MediaWiki is not needed to be private, but the user can decide if anybody else should have access to the MediaWiki.
MediaWiki does not provide a client that should be installed on each computer, as it should only be installed on the server. It is web application based where the user can upload files via a browser if wanted and this can also be used when looking at a files revisions.
More users can upload to the same MediaWiki stored on a web server, in fact the idea is that the user does not even need to trust the host of the MediaWiki. MediaWiki does not handle conflicts in files very well, as the MediaWiki is not build with the idea of only using this as storing files. This is all handled by revisions which means that if there is a conflict it is not lost, but is stored in a separate file (older revision).

2.5.2 MediaWiki design

Finding a backend for storing files can be difficult, and in the description it could sound like MediaWiki is all good. This is obviously not the case as there are downsides with it. As mentioned it is not directly build for using as a remote storage for files only. This becomes evident when looking at the design, as the structure is flat. It is not possible to upload directives to MediaWiki, unless one would do it with some compression method like zip or rar. When downloading files from a wiki, it can be done simply by accessing the direct link to the file. Each file is uploaded as it would be a picture for a Wikipedia site and this could give troubles when creating a complete file system like DropBox, but it does not make it impossible.
A thing to mention about a Wiki is that it is not designed to keep some things a secret. However, when first having access, the idéa is that you should at least have read access to all of the content. This is no problem in this project, because all the files stored in the MediaWiki have been encrypted by the client.
As the software code is open, each MediaWiki could be designed completely as the administrator would want, but this is not what is supposed to happen. However, the Wikimedia foundation made it possible to do small tweaks only for that hosted MediaWiki, without ruining all the basic principles of a Medi-

aWiki. This is done trough a LocalSettings.php files, which is stored on every MediaWiki. It is possible to enable an API to use for the MediaWiki, and this will make it easy to use programming wise. The MediaWiki API is a REST web service and the commands are HTTP Request, such as GET, PUT, POST, DELETE.

2.5.3 MediaWiki API

As mentioned the MediaWiki offers a great API[Medd], which is very easy to use, just having a little knowledge on how APIs work. The API is open for everyone to see as well, and even if wanted, one could develop their hosted MediaWiki to have a bigger API than the original. The MediaWiki API is a web service that provides all the features, data, meta-data etc. over HTTP.
The API offers a lot of different commands:

- Upload Image Info (file)
- Download file
- File content
- Revision file URLs
- Log in (if it is not an open MediaWiki)
- Log out

Upload Image
As mentioned MediaWiki is not build to be a file system, but can do that with no problems. The MediaWiki are considering the files as pictures (hence the image name), and not all file extensions are accepted as default, which is why the user needs to specify the file extensions into their own MediaWiki.
The command for uploading is build up the following way[Medc]:

<link_to_the_MediaWiki>/api.php?action=upload&filename=<the_filename>
&url=<url_to_the_file_on_MediaWiki>&token=<the_token_provided>

This is an example on how to upload the file using the MediaWiki API. Other parameters can be used as well. In the example from above, the token is given after a successful login to the MediaWiki, meaning it is not necessary when the MediaWiki is open.
Uploading is needed to be enabled on the MediaWiki before it works.

Downloading File
This is not really done by using the MediaWiki API, but is done simply by direct linking to the file on the MediaWiki. Getting the direct link to the file in

the other hand, can be hard if it was not for the API. The command for this is simple [Meda]:

```
<link_to_the_MediaWiki>/api.php?action=query&titles=File:<the_filename>
&prop=imageinfo&iiprop=url&format=xml
```

This command will output XML that contains the direct url (because of the iiprol=url parameter). Having the direct link to the file it can be downloaded easily. Other parameters can also be used to get more file information if necessary.

File content
The file content can be used to get the content uploaded next to a given file. The idea here is to use the content as a check up with the file on the current system. There might have been an overwriting which none was interested in, and possibly more, which means it can be useful to get older revisions to get the old one back. This is done by the following command[Medb]:

```
<link_to_the_MediaWiki>/api.php?action=query&titles=File:<the_filename>
&prop=revisions&rvprop=content&rvlimit=max&format=xml
```

Outputting the XML, giving file content for each revision. Other parameters are available too.
The iilimit=max parameter, is for not only getting one file but getting the maximum of files per page (currently this is only 500). If there are more files than 500 iicontinue has to be used.

Revision urls
Revisions are needed in this project, as the user might be interested in going back to an older revision. This is done using the imageinfo command from earlier, but with a little change in parameters. It looks like this:

```
<link_to_the_MediaWiki>/api.php?action=query&titles=File:<the_filename>
&prop=imageinfo&iiprop=url&iilimit=max&format=xml
```

Outputting the XML, including the urls for the revisions.

Log in
This part is only needed if the MediaWiki is not an open MediaWiki, and the command is very simple (this has to be a POST command):

```
<link_to_the_MediaWiki>/api.php?action=login&lgname=<username>
&lgpassword=<password>&format=xml
```

This will return a cookie in the HTTP header and a confirmation like 'Success' or 'NeedToken'. If it is the case that it returns 'NeedToken' another call have to be made just after:

<link_to_the_MediaWiki>/api.php?action=login&lgname=<username>
&lgpassword=<password>&lgtoken=<the_token_provided>&format=xml

The user should now have a cookie, that can be used for the earlier mentioned methods. If the user has access on the MediaWiki to these.

Log out
As well as with the log in, log out is only needed if the MediaWiki is not an open MediaWiki and the user wants to sign out:

<link_to_the_MediaWiki>/api.php?action=logout

This will delete the log in token provided earlier and browser cookies. For security measures this should be called every time the user is done calling the MediaWiki to be sure there are no valid tokens that can be used for the specific user.

2.5.4 Revisions

The MediaWiki stores all revisions for a file and this is one of the reasons why it is good for the project. However, as it is not a system that is directly build for file storage, it lacks a few operations which are provided by remote file storage systems.
There is no such thing as conflict handling, meaning that if two users changing in the same file at the same time, the last one to upload the file, will basically have uploaded the final file. However, none of the other uploads will be lost, as they will be stored as an earlier revision. Conflict handling could be a nice tool to have, but it is not needed in this project.

2.5.5 Other Storage possibilities

This section will only give a small view on which other systems could be in consideration for remote file storage solution.

Django Custom File Storage[dja]
The first one is a Django web host, custom file storage library. Django is a

good web hosting solution and using it as a storage would not be bad. However, revisions are missing in this, as it is used for when actually hosting a web page, so this should be developed by the user. Django is written in Python which offers a lot of libraries for other possibilities.

OFS File Storage[ofs]
Yet again another Python storage possibility, which as with Django would give the possibility to offer a lot of python libraries. OFS File Storage offers setting up an easy to use web api, but as with Django there is not a native revision handling methods, so this should be developed. There is a lot of nice methods included in this library.

RemoteStorage.io[rem]
Remote Storage looks similar to already existing file host solutions, and this would not be a clean file storage. Developing own methods is possible though as it is open source. This looks similar to Git or SVN but more having complete files in mind.

Git[git]
One possibility could be to use Git as it is the new hot subject in version controlling. It is free and open source and is able to store files very easily. Git is famous for the conflict handling and the merge possibilities and is capable to store large projects with no performance issues. It handles revisions very well as it is a version control system and it saves revisions all the way back to the initialization of the project. Branches are also available in git and this is great when more people have different suggesting in changing a file.

2.6 Existing Encrypted File Systems

Many file systems already exists with all kinds of different purposes. The idéa of this section is to find and look at some different file systems, as in this project a file system needs to be developed. It does not make sense to just look at random file systems, but to have a look at file systems with a more cryptography point of view.

2.6.1 Cryptographic File System

When looking into Cryptographic File Systems, the first that comes to mind is the Cryptographic File System for Unix[Bla93]. This system is kind of a classic,

when it comes to cryptographic file systems, and might not use the newest technologies, when it comes to cryptography.
It is one of the first, to have encryption directly as a part of the file system and the idea is that the user have a cryptography key, and a directory they want to use.
The encryption / decryption happens as the user is working, meaning that documents containing plain text are never actually stored on disk. It is especially not stored remotely as plain text, but is always encrypted.
The CFS uses DES as the encryption method, and according to the article [Bla93] more modes of operations are available. The one described is a mixture of OFB and ECB mode, making sure the pattern recognition problem with ECB is avoided.

2.6.2 Transparent cryptography File System

The TCFS[Mau] was developed at Dipartimento di Informatica ed Applicazione of the Universita di Salerno in Italy and back in 1997 [CP97] and can be seen as an extended version of the NFS. NFS has a hard trust model, meaning that when storing data remotely using NFS, it is assumed that one trusts the server. This is not always the case and TCFS is better at that point. Instead of trusting the server TCFS will allow the user to encrypt files them self before storing them remotely.
TCFS as well as CFS uses DES as encryption algorithm, but can use other block ciphers, and stores keys in an database on the system. The keys are encrypted by the user's master password meaning they are never stored locally in clean text. The idea behind TCFS is that they want to keep the trust level to a minimum.

2.6.3 Encrypting File System

The Encryption File System[Wri] (EFS) is a Windows built in function used to encrypt and decrypt files and is a part of NTFS. It was initially designed due to the lack of security in the NTFS as it was easy to bypass permissions needed in the NTFS.
EFS uses a mix of symmetric and asymmetric-key encryption and the first time a file is encrypted the EFS will give a public- and private-key pair for the users account. After this it generates a random number which is called a File Encryption Key (FEK) and this will be used to encrypt files with DESX [Rog] which is an extended version of DES processing the data three times with three different keys. Then after this the FEK is stored encrypted using the public-key.

The public- / private-key encryption algorithm is RSA.
The way of using symmetric-keys for the files and then public- / private-keys for the symmetric-key is a great way to give good performance while not missing out on the security. Public- / private-key encryption schemes are not good to use for huge amount of data as they require way more performance to do so (see section 2.1.2.
EFS have some flaws which the biggest one is the use of the recovery agent and this uses a recovery key which is a second generated key that the administrator of the system can use to recover data. It is smart for big companies as they never know what happens to their employees, however it is also a major security flaw. The problem is that in a typical default setup the local system administrator is chosen as being possible to be the recover administrator and for most regular users in windows it is easy to get this access as the password for this user is usually blank. This is a whole other subject, but needed to be mentioned as well.

2.6.4 eCryptFS

The eCryptFS is a kernel-native stacked cryptographic file system used on Linux[Hal]. This means that the file system is layered on top of already existing mounted file systems. The way eCryptFS works is that it encrypts / decrypts the files as they are written or read to and from the lower file system. The idea is that applications working in the user space, use file system calls that goes trough the Virtual File System. eCryptFS works mostly in the kernel, however on top of that it might do some key management from an eCrypt daemon running in the user space. It should mostly work in the kernel by getting keys from the users key ring and only use the cryptography api from the kernel to do all the encryption and decryption. Key rings are stored in the kernel as well. As a default eCryptFS uses AES as the cipher, but the user can basically use all the ciphers available in the kernel.
eCryptFS will only protect the confidentiality of the data, which means that if one wants to use eCrypt they should use some kind of access control on a trusted host.

2.6.5 CryptoFS

CryptoFS is an encryption file system using FUSE and Linux Userland FileSystem[Hoh]. CryptoFS uses a regular directory on the computer to store all files encrypted and then the mount point will have all the decrypted files. If the user unmounts, the system needs to mount again using the users password to be able to access

the files again.
When mounting the system CryptoFS generates a key specifically for the requested cipher using the requested message digest function. The different algorithms have their own required key size. When the key have been generated salts and initialization vectors are generated (if these are needed) and will be used in the encryption later.
CryptoFS seems like a good basic encryption file system if a user wants to ensure their data is encrypted.

2.7 FUSE

FUSE or Filesystem in Userspace, is a UNIX software interface and it can be used by users to have their own file system without accessing the kernel interface. However, it is a bridge to the kernel interface, accessed from the user space.
This section will look into FUSE as it is essential to look how FUSE works to fully understand it. When looking at FileSystem in Userspace, the first thing to understand is the difference between kernel- and user space. Then why a FileSystem is good to have in Userspace and in the end what kind of methods FUSE provides.

2.7.1 Kernel- and User Space

There is a big difference between kernel- and user space. The kernel space is somewhat everything that is being done within the kernel of a system. This also means that changing the kernel space or code, means that one needs to have special rights to do so, as not everyone should be allowed to change what is in the kernel space. In fact most people should not be allowed to do so.
Jeffrey Layton uses a great example [Mag] of a programmer that wants to create something, and this something could be developed into the kernel. However, this can be dangerous as if there is a bug, it could potentially create a bug for the whole system. When moving the code or application to user space, this is never gonna be a problem as the kernel will only provide the necessary resources, but not actually affect the kernel it self. This also means that the kernel will remain stable when testing the application. In fact this means that the application can get to test faster, as a bug in the application only means it crashes the application it self and not the kernel.
So staying in user space for a file system is perfect, as if it crashes, one could

easily just end the started processes and remount the file system again, and never be in danger of an entire system crash.

2.7.2 FUSE Introduction

Creating a File System for specific purposes, was impossible to do without interacting with the kernel space. It is possible now, as the FUSE library have been created. It is a great tool for writing a file system that works in user space and it includes a very good API to do so.
The way FUSE works around the scenario of working in the kernel space, is through a Virtual File System (also called VFS), as it need help from the kernel to work. The VFS is located within the kernel but is accessible for users only working in the user space.
 Looking at figure 2.5, which is the hello.c example from the FUSE Github

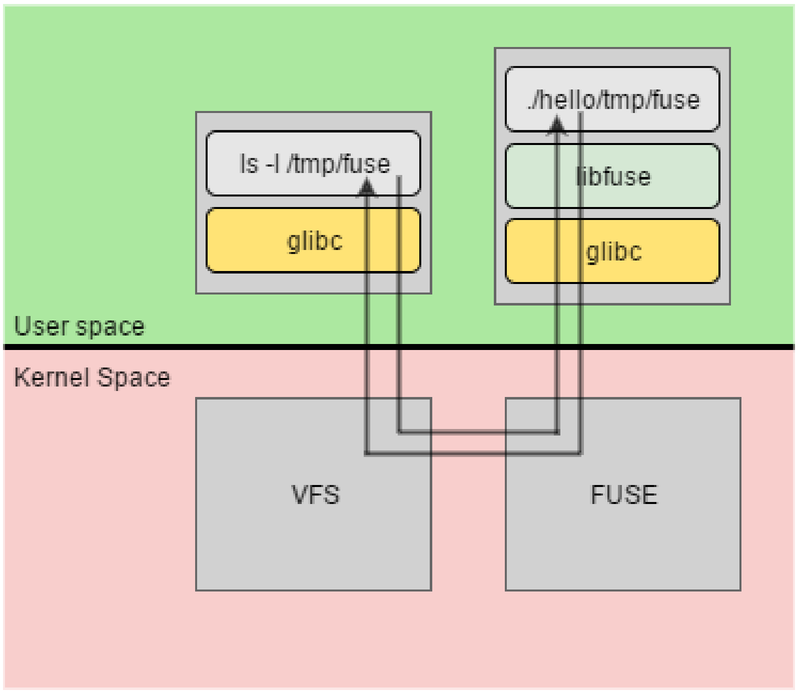


Figure 2.5: The general flow of a hello world file system using FUSE.

Repository, it creates a hello world binary that is mounted at another mountpoint. The way it works is that creating the mount point (upper left), it works using the glibc through VFS and into the FUSE in the kernel, back through glibc and then to the libfuse (which is the FUSE library in user space) to get the binary hello world. Here the hello world is compiled and it goes all the way back to the mount point. It can be seen that the user never works directly in the kernel space at all, and this is one of the keys to why FUSE is so useful.

2.7.3 Methods in FUSE

The FUSE API includes a lot of different methods or operations that can be used. IBM provides a good introduction[SS], but also a good explanation of these methods.

- **getattr** is used to retrieve the extended attributes.
- **readlink** reads the target of a symbolic link.
- **symlink** creates a symbolic link.
- **link** creates a hard link to a file.
- **getdir** reads the content of a directory.
- **mknod** creates a file node.
- **mkdir** and **rmdir** creates and removes directories.
- **chmod** changes the file permission bits.
- **chown** changes the user/group ownser ship of a file.
- **truncate** changes the size of a file.
- **utime** is used in the modification of the time in a file.
- **open** opens the file (it does not read the file).
- **read** reads the data from an open file. Returns the number of byte requested.
- **write** writes data to an open file.
- **close** closes an open file.
- **statfs** gets the statistics of the file system.
- **flush** is representing the flush-cached data.
- **release** is called when there is no more references to an open file, descriptors are closed and all mappings are unmapped.
- **setxattr**, **getxattr**, **listxattr** and **removexattr**, set, get, list and remove extended attributes.

This is all the methods/operations that the FUSE API offers[SS], when creating a file system.

2.7.4 FUSE-JNA

The FUSE API is written in C, however it is also possible to work with FUSE in other programming languages. The beauty of this is that other languages than C are providing first of all easier implementation, but also tons of libraries that can be a great benefit when developing a file system.
Especially the FUSE-JNA[Eti] makes it easy to develop a FUSE file system, without the need to define all FUSE methods, and structures.

2.8 Key storage

When storing keys and key rings there is a need for a format or system to store these in. This could be done with one of the following:

- Online database
- Local database
- Markup language

The important thing in this project when selecting the storage option for the keys and key rings is that they should be easy to add and retrieve. It should also be possible for users to share keys between each other and easily update them without delete and/or ruin already uploaded keys. This can be done by having revisions stored to make sure if a key is ruined that an older version can be read.

2.8.1 Online databases

An online database could be one way of storing keys and key rings. Most databases are made for storing a lot of data and retrieving the data fast by having it indexed. Besides that the online database needs to be hosted some place and needs to be accessed from some where else. This can be a security risk because now there is one more system containing data, even though it is encrypted. This would require more work for the end user because they need to set up a database and the user permissions.

2.8.2 Local database

A local database is almost like an online database except it is running on a local machine. By having it locally and without it being accessed from the outside it reduces the risk of an intruder. The problem here is synchronizing keys and key rings between users. This is a problem because users depend on each other when reading and/or updating shared keys.

2.8.3 Markup language

A markup language is not a system like a database but a document that stores data in a format that is both easily readable by humans and machines. The thought behind having a markup language storing the keys and key rings is to have files containing them and later uploading the key or key ring to the online file storage. This is still accessible from somewhere else but at least it is not another system to maintain. This stores the keys and key rings like they were files and just be downloaded and uploaded like every other file.

2.9 Summary

In the theory chapter different subjects have been researched and these subjects are: general cryptography, access control, key rings, key sharing, MediaWiki, file systems and key storage.

The general cryptography section looked into the difference between symmetric- and asymmetric cryptography as well as looking at different already used cryptography schemes. Hashing was also looked at in this section. Access control was the next section where different used access controls variants where looked at. Next thing was to look at key rings as well as the key sharing included finding an idea for key rings and also how PGP works.
MediaWiki was looked into as well as this is used as the server / backend for the remote file storage. Researching different kind of cryptographic file systems that exists already, and possibly got some inspiration from these. Many of these use FUSE and this project is supposed to do so as well. In the end looking at how to store the keys using online- or local database or a markup language such as XML.

Chapter 3

Requirements

The requirement chapter's goals are to define the needs and requirements of the application. The chapter starts with a brief description of the application, followed by some use cases. Afterwards the cryptographic needs are found and then the more technical requirements to the system are found. This should lead to a more clearer view of what the application actually contains.

3.1 The Application

Looking at already done implementations from earlier, there are a lot of possibilities on what this system could be. The idea of the project is to have a file system on the computer where the user can work with the files and then remotely store them. When the user wants to store the files remotely the server is not trusted.
Given this the whole project can be divided into several categories which are:

- A file system handling all files stored locally on the computer. There is no need to encrypt files while working, however when done working with the files they may never be stored locally in plain text. They could either be uploaded and deleted, for then to be downloaded again when running the

application, or stored encrypted after use and decrypted next time when running the application.
- The local storage should be a FUSE application meaning it is only required to work on Linux distributions.
- A remote file storage is needed to be implemented as the user needs to synchronize all files at all time. MediaWiki will be setup to do this as it is very easy and revisions is a possibility. Conflicts and merging is not a requirement to the current system, but revisions should be implemented. Trusting the server is not needed so the MediaWiki can be open if that is what a user want.
- Files stored remotely needs to be encrypted as someone might be able to open the files, however not everybody should be able to read files.
- Key management is needed, as a user should be able to share access to a file with another user.

These points sums up the basic needs of the system being a FUSE local file system, with MediaWiki as remote storage and some kind of key management.

3.2 Use Cases

The system have various needs, which can be difficult to discover right away and in order to do this, some use cases have been developed. These will later be developed as tests that also can be used to check if various functions are working as intended.
A thing that is needed to be known to fully understand the use cases is that there exist keys and key rings, where a key ring is simply just a collection of keys.

3.2.1 Initialization of the Application

This is the basic use case when the user starts the system.

1. User initialize to start the application.
2. Application reads a config file from the user including; MediaWiki setup (url, file extension for encrypted files and optional username and password), root directory and a key k.
3. Application downloads the belonging key ring kr and its content using k.
4. Application decrypts content c using read-key in k.

5. Application validates that the encrypted kr's checksum is equal to c.
6. If kr is not valid the next revision is downloaded, validated until it is valid.
7. When kr is valid it is decrypted using the symmetric-key and IV in k.
8. Runs through all keys in kr and download and decrypt all corresponding files and key rings from kr.
9. When there is a new key ring it downloads, validates and decrypts all files and possibly new key rings. Continues until there are no more key rings to run through.
10. Mounts the file system including all the downloaded files.

The user can now change the files as wanted, and further use cases can be developed from this.

3.2.2 Uploading a new file

Uploading a file is essential to this application, as it is the most basic usage other than initializing the system. It is assumed from this use case that the system have been initialized.

1. User creates a file f in the mounted folder.
2. Application creates a key k pointing to f.
3. Encrypts f using k.
4. Generates a checksum c for encrypted f and encrypts it using write-key in k.
5. Uploads both encrypted c and f to the MediaWiki.
6. k is added to the parent key ring kr.
7. Parent key pk for kr is then found.
8. Encrypts kr using pk.
9. Generates a checksum c2 for encrypted kr and encrypts it using write-key in pk.
10. Uploads both encrypted c2 and kr.

After creating and uploading the file, the user might want to change this file.

3.2.3 Updating an existing file

The user wants to change a file, that already exists. It is already a part of the mounted system. It is assumed that the user has created a new file, downloaded, verified and decrypted the corresponding key for the file and initialized the system.

1. User changes file f, in the mounted directory and saves f.
2. The application looks up key k pointing to f.
3. Encrypts f using symmetric-key and IV in k.
4. Generates a checksum c for encrypted f and encrypts it using write-key in k.
5. Uploads both encrypted c and f to the MediaWiki.

3.2.4 Change file name

It happens that the user only wants to change the file name of a file, not necessarily the file data. It is assumed that the user has a file in the mounted system, downloaded, verified and decrypted the corresponding key for the file and initialized the system.

1. User changes the file name of file f. The old file name is considered o_fn, new is n_fn.
2. Application finds the key k pointing to f.
3. Updates name in k from o_fn to n_fn.
4. Application finds the key ring kr containing k.
5. Application finds the key pk pointing to kr.
6. Encrypts kr using pk.
7. Generates a checksum c for encrypted kr and encrypts it using write-key in pk.
8. Uploads both encrypted c and kr.

3.2.5 Download file

The user might want to download a file, after having uploaded some file. It is assumed that the system has been initialized, a file have been uploaded to the MediaWiki and the user has the key for the file.

1. Application finds and download f and its content using k.
2. Application decrypts content c using read-key in k.
3. Application validates that the encrypted file f's checksum is equal to c.
4. If this is not valid the next revision is downloaded, validated until it is confirmed being the right file.

5. When f is valid it is decrypted using the symmetric-key and IV in k.

This is done when the user wants the most updated version of the file, however this is not always the case.

3.2.6 Downloading older revision of File

The user might want to download a file, but not necessarily the newest version of the file. Downloading an older revision of the file is essential to this application. It is assumed that the application has been initialized, and there has been uploaded several versions of the file to the MediaWiki.
It is important to notice that each revision are ordered by their upload date in descending order in the application. This gives the latest uploads first.
This is one of the essential things behind MediaWiki, because it makes sure other users can not screw files up by encrypting them wrong, delete file content, etc.

1. Application finds the file f by using key k.
2. Finds and download f and its content using k.
3. Decrypts content c using read-key in k.
4. Validates that the encrypted file f's checksum is equal to c.
5. If is not valid the next revision is downloaded, validated until it is confirmed being the right file.
6. When f is valid it is decrypted using the symmetric-key and IV in k.

3.2.7 Sharing File with Read & Write access

When the system is up and running and the user has uploaded and downloaded files, they sooner or later want to share some of their files. This is done by sharing the key to the file in between them. A user has the option to allow other users to get full or only read permission as long as the user himself have these permissions. That means a user with only read permission can only share the read permission to another user, because he is not in a possession of a write-key. It is also possible for a user to share a key belonging to a key ring that contain multiple keys. The connection between the users are not covered in this project and is assumed to be a secure line.
In this use case it is assumed that more users exist, the system has been initialized, a file already exists and the file is uploaded to the MediaWiki and the key

for the file is downloaded, verified and decrypted. It is also assumed that both users can connect to the same MediaWiki.

1. User B wants to share file f with user A.
2. User A generates a public-key pub and a private-key piv.
3. User A sends pub to user B (assuming the line between them is secure).
4. User B finds belonging key k for f.
5. User B generates a secret s.
6. User B clones k2 and keeps read-key and write-key in it.
7. User B encrypts k2 by using s.
8. User B encrypts s by using pub.
9. User B sends encrypted k2 and encrypted s to user A.
10. User A decrypts the encrypted s by using piv.
11. User A decrypts the encrypted k2 using decrypted s.
12. User A adds decrypted k2 to a key ring kr.
13. User A finds the key pk pointing to kr.
14. User A encrypts kr, with the new key k included, using symmetric-key and IV in pk.
15. Generates a checksum c for encrypted kr and encrypts it using write-key in pk.
16. Uploads both encrypted c and kr.
17. f can now be downloaded, verified, decrypted and written, encrypted and uploaded.

User B can now change and upload new revisions of the file.

3.2.8 Sharing File Only with Read access

It is also a possibility that a user only wants to share a key, but only giving read access. It is assumed that the first user has read access already. Connection is as mentioned not covered in this project, and this means it is assumed to be shared on a secure line.
In this use case it is assumed that more users exists, that the system have been started and that the file already exist and is uploaded to the MediaWiki.

1. User B wants to share file f with user A.
2. User A generates a public-key pub and a private-key piv.
3. User A sends pub to user B (assuming the line between them is secure).
4. User B finds belonging key k for f.
5. User B generates a secret s.

6. User B clones k2 and only keeps read-key in it.
7. User B encrypts k2 by using s.
8. User B encrypts s by using pub.
9. User B sends encrypted k2 and encrypted s to user A.
10. User A decrypts the encrypted s by using piv.
11. User A decrypts the encrypted k2 using decrypted s.
12. User A adds decrypted k2 to a key ring kr.
13. User A finds the key pk pointing to kr.
14. User A encrypts kr, with the new key k included, using symmetric-key and IV in pk.
15. Generates a checksum c for encrypted kr and encrypts it using write-key in pk.
16. Uploads both encrypted c and kr.
17. f can now be downloaded, verified, decrypted.

User B can now read the file, and is able to change the file locally. Technically B can create another write-key and upload the file, but all the other users with the read-key for the file would be able to detect an invalid revision. The other users would then just ignore the invalid file and download the latest valid file available from revisions in the MediaWiki.

3.2.9 Ending the session

A lot of operations or methods are now covered, but closing the application is also essential to a file system, as it cannot be closed as any other application. It is assumed that the system has been initialized and there are files created and downloaded in the mounted system.

1. User wants to close the file system application.
2. User gets to choose if the user wants to upload all files in the directory to the MediaWiki.
3. If no: All files are deleted locally and program is terminated.
4. If yes: Application runs through the all the keys.
5. For any key k, locally stored, file f is found.
6. Encrypts f using symmetric-key and IV in k.
7. Generates a checksum c for encrypted f and encrypts it using write-key in k.
8. Uploads both encrypted c and f to the MediaWiki.
9. If k is a key ring, it runs through this and do the same for all the keys. However, for each child key rings the parent key pk is found and used to

encrypt and upload the key ring as well.

10. In the end reaching the root key means all files and key rings have been uploaded.

In the user case above, it is stated that the files are deleted locally, this could also be possible to have them stored locally, as long as they are encrypted, because other users could get access to the files.

3.3 Cryptographic Requirements

Cryptography is an important aspect in this project, because it only allows users with the secret, to the data, to read or write to it. Having the only users with the secret to read and/or write the data, is not the only thing being apart of the cryptographic needs. There are some things needed for this project to be considered and this section will include part of this.
Confidentiality, integrity, availability are three important components describing a secure system, called the CIA triad. The CIA concepts are the fundamental concepts in security. The CIA concepts are not the only ones needed for this project, as other important elements are missing; such as authorization and authenticity.

3.3.1 Confidentiality

Confidentiality or secrecy is to protect the information from disclosure to unauthorized parties. This can be done by encrypting the information to make sure only the people with the secret can read it. Confidentiality is a big part of this project and is, as already mentioned, a part of the CIA. When encrypting files using symmetric cryptographic schemes, it protects the files by encrypting all the files in the system. This makes it only readable and writeable for users knowing about the secret for decrypting the file and this gives confidentiality.
When having an open MediaWiki a third party can get a hold onto the files, but as everything are encrypted the confidentiality is ensured to a high enough level.

3.3.2 Integrity

Integrity is to protect the information from being modified by unauthorized parties. This can be done by including a hash of the information sent and comparing it with the hash of the original message. It is difficult to protect unauthorized people from modifying the data on an open MediaWiki if the user wants that. However, an such unauthorized change can be detected, and along with revisions the user can get back to the last authorized version of a file. This can be used to ensure the integrity of the data.

3.3.3 Availability

Availability is to ensure that authorized parties are able to access the information when needed. This can be done by having an off-site backup location of the information to limit the damages caused by hard drive failure, DoS attacks, natural disasters and more.
For an open MediaWiki it can be hard to ensure availability all the time, as the mentioned DoS attack. The connection between the client and the MediaWiki can easily be blocked, or another way is that if an attacker ever gets a hold of the files encrypted name on the MediaWiki, they can just spam it with new files with the same name. The user will be able to detect this through the integrity, but it can be a major performance issue to get through all revisions to find a right one, if say the attacker have spammed a thousands of new revisions to that exact file.
It is difficult to avoid DoS attack in this kind of setup, but it is somewhat covered by using revisions, performance wise it will be difficult to cover completely, meaning the performance issue will not be within the scope of the project. Only that the user in theory is able to recover the data.

3.3.4 Authorization

Authorization ensures that the user requesting some kind of access to the files, are in fact having that kind of access. A user with read access only, should not be able to write to the file. As mentioned in the integrity section 3.3.2 it is difficult to completely stop unauthorized actions, however they can be detected. If a user only has read access, they only get the read-key and the symmetric-key to the file. This means they can verify the file using the read-key, but when they want to write and do not have the write-key, another user with read access can invalidate the files an go on to the next revision. This means the authorization is possible to maintain, but this can give performance issues.

3.3.5 Authenticity

Authenticity is to ensure the data, transactions, communications and so on are genuine. It is also to validate that both parties involved are who they claim to be. It is not needed in this project to authenticate the server, as the whole idea is to keep the trust level down for the MediaWiki server. Authenticity can be needed when sharing keys in between the users, a write-key in the wrong hands, can be a major disaster. Authentication can be done by having features such as digital signatures.

3.4 System requirements

The cryptographic requirements are not the only one needed in this project, as some certain functionality is needed as well. The system requirement section is to cover what kind of requirements that are not regarding cryptography.
It will be divided into three parts the client system, key handling and MediaWiki. The key handling is in principle also a part of the client system, but they are separated in this section to simplify things.

3.4.1 Client system

The client system is basically whenever the user is changing files locally a operation is needed in order to do so. This subsection will not cover anything regarding the key handling section.

3.4.2 Initialization of system

When initializing the system, the user should either have a key stored or input a master password. When doing so the system should create a key type using the master password. Now after this is done it should download the users corresponding key ring from the MediaWiki and decrypt it. If it does not exist it means the user does not have one and a new one should be created.
Download the key ring, decrypt it and start downloading the files this key ring is

linked up to. If there are other key rings included, they need to be downloaded, verified and decrypted as well. This stops when a key contains a file (not a key ring, but a file the user has uploaded). This file is then downloaded, verified and decrypted.

3.4.3 Creation of a file

An essential part whenever start working in a file system is to create a file, initially or at any point. What needed to be done is that a key for this file should be created and stored in the corresponding key ring. There are some requirements to this key, as one simple key is not enough. The key need to include a read- and write-key as well as a symmetric-key, IV and name of the file after the decryption.
When creating a file the system needs to encrypt the file using the symmetric-key and IV. After the file has been encrypted there should be generated a file hash (checksum). This file hash needs to be encrypted with the write-key. Already when creating a file, the system needs to upload the file and file hash to the MediaWiki.
Regarding the key, it is added to a key ring, and this key ring's parent key needs to be found, and used for encrypting and uploading the updated key ring.

3.4.3.1 Upload file

The user change files all the time, and whenever there is a change to a file, this file needs to be uploaded to the MediaWiki. It is expected during an upload to the MediaWiki that the system find the right key. The key contains a read- and write-key, symmetric-key, IV and what the name for the file should be after it has been decrypted. The symmetric-key and IV in the key is needed for encrypting the file. After the file has been encrypted there should be generated a file hash (checksum). This file hash needs to be encrypted with the write-key.

3.4.3.2 Change file name

The user might want to change the name of a file, without changing the content. The file name is depended on the information from the key. This means when a file name is changed, the system needs to find the key accordingly, change the name entity. Find the parent key of this key, encrypt the file, its file hash and finally upload the key.

As the key is included in a key ring, it should be updated in that key ring. The key ring needs to be encrypted using its parent key and then the key ring can be encrypted, generate file hash, encrypts file hash and finally upload the key ring.

3.4.3.3 Download file

When downloading a file, it is assumed that the initial key ring already have been downloaded and decrypted. So needed from the system is to find the corresponding key, which might require downloading new key rings and decrypt these. After the key for the file is found, the hash for the encrypted file should be downloaded and verified using the read-key. When it is verified the file should be downloaded and decrypted using the symmetric-key.
If the file is not verified the next in line revision should be tried until a revision is verified, downloaded and decrypted using the symmetric-key and IV.

3.4.3.4 Download revision

A user might want to download an older revision of a file which the system should be able to handle this. When a user requests an older revision this should be downloaded, verified using the read-key and then decrypted using the symmetric-key and IV.

3.4.3.5 Deletion of a file

When a user wants to delete a file, the file should be removed from the user's local file system. The file is never removed from the MediaWiki. The key ring containing the key for the file has to be changed by the user by having the user removing the key. This gives a key ring modified to contain all the same keys but without the key for the file the user just removed locally.
After the key ring has been modified the key ring can be uploaded by encrypting the key ring and its hash. This requires the user to have write permission.

3.4.3.6 Read- / Write-key

As mentioned there is needed to be a read- and/or write-key included in each key. The write-key is used to encrypt the file hash of the encrypted file, and the read-key is used to verify this file hash when downloading.

3.4.3.7 Working File System

The user wants to work in a file system, and do not want to manually handle all the operations mentioned above. This means there should be a working file system that automatically handles; upload, download and change file name, while the user is working.
When a user wants to download an older revision of a file, the user should call this method manually and it is not a requirement to the system to do so automatically.

3.4.4 Key Handling

The key handling part is done by the client system as well, however for the simplicity this is separated in the requirement section. The methods for the key handling are; creation, verifying, sharing and removal of sharing.

3.4.4.1 Creation

When creating a key a couple of attributes should be included; a newly generated key id, symmetric-key, IV, read- and write-key, MediaWiki information and a name for the file when it is decrypted.
When all attributes have been set, the key should be added to the corresponding key ring. The parent key of the key ring should be found and used to encrypt and upload the key ring and its file hash to the MediaWiki.
It should be defined in every key whether it links to a key ring or a file by an attribute.

3.4.4.2 Sharing

When sharing a key from user A to user B, the key needs to be added to user B's key ring. User A has the option to add the same or less permissions as user A already has. This gives a user with read- and write permissions the availability to share the file with only read permission or with both read- and write permissions. A user that only have read permission can only share the key with read permission. User B chooses which key ring it should be added to.

The corresponding parent key to that key ring then needs to be found and used to encrypt and upload the updated key ring to the MediaWiki.

3.4.4.3 Share removal

Removing a shared key cannot be done, as a user should not be able to remove keys from another user. An option for this to happen is to copy all shared files and include them in a new key ring. This key ring and files are then encrypted and uploaded to the MediaWiki with a new parent key. This key ring and files are now only readable and writeable by the user who uploaded this. It is of course possible for the user to share these just like any other key ring or file.

3.4.4.4 Update file name

This method is whenever a user decides to update a file name. The file name attribute in the corresponding key should then be updated. Parent key should be found, used to encrypt and upload the updated key.

3.4.5 MediaWiki requirements

The MediaWiki has been chosen as the remote file storage in the cloud, and can be consider the server side of the project. It has some requirements as well that are covered in this section.

3.4.5.1 Setup requirements

Using a default MediaWiki setup is not enough, as some things are disabled.

1. API needs to be enabled
2. All, non default, file extensions needs to be added as accepted file extensions.
3. Uploading needs to be enabled.

When these are done, the MediaWiki setup should be as intended.

3.4.5.2 Storing of Files

The MediaWiki should be able to store the encrypted file types. It is not a requirement that all file types are needed to be stored, but only a specific encrypted file type. A requirement for storing files is that no one should be able to delete files.

3.4.5.3 Storing of Keys

The MediaWiki should be able to store encrypted key rings containing keys.

3.4.5.4 Revisions

Revision is a huge part of this project, and as the user should be able to download a revision of a file through the client system, MediaWiki needs to support this as well.

3.4.5.5 File Content

During the encryption of a file, the hash of the file needs to be encrypted using the write-key. When downloading a file this is used to verify that the uploader had authorization to change the file.
MediaWiki needs to support some extra file content to be uploaded a long with a file, that can be used for this encryption of the hash.

3.5 Summary

The requirement section was made to specify the different requirements for both the application but also the cryptographic needs. This was needed before being able to design the actual application / system. Firstly finding the general overview of what the application actually is supposed to be, next thing creating use cases to specify what kind of usage there could be on this system.
Cryptographic requirements was also specified using the CIA and more and in the end the non cryptographic system requirements was specified.

Chapter 4

Design

In the design chapter, the purpose is to get an overview of what needs to be implemented. This will start out by the overall design, that defines the needed libraries. Use cases are defined in the requirements section 3, and here they will be specified in a more technical manner using sequence diagrams. In the end there will be a need for the cryptography choices done in this project.

4.1 Overall System Design

Showing the flow of the program is done in the integration section 4.3, but first an overview of the system is needed to be designed. Using the requirements defined earlier, the needed classes / libraries have been identified and can be seen in figure 4.1.

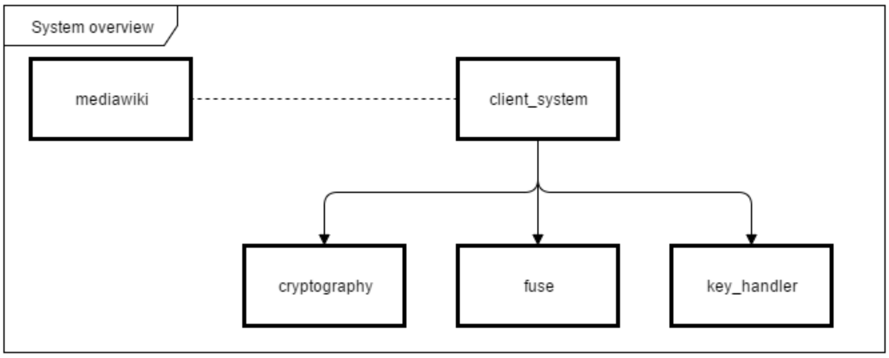


Figure 4.1: Overview of the system design, including the identified needed libraries.

Looking at figure 4.1 it can be seen that the following libraries are needed:

- **client_system**; the client that the user performs to work in.
- **key_handler**; all the key operations handled in the client_system, such as creating keys and key rings, as well as finding and adding keys to key rings.
- **cryptography**; all the cryptography operations handled in the client_system, both asymmetric and symmetric as well as hashing operations.
- **FUSE**; all the FUSE or file system operations, handled locally on the client. This includes operations such as create-, rename-, change-, delete file and more.
- **MediaWiki**; the operations to and from the MediaWiki, such as upload and downloading files.

It can be seen that most operations in this system, is done in the client_system. Splitting it up into three main libraries and a communication library with the MediaWiki. These are the five libraries used in the integration part as well.

4.2 Key handling

Key handling is done by storing keys in files using a markup language. A file can contain many keys, which is called a key ring. These key rings can then be uploaded to the MediaWiki as they are files.
That means the MediaWiki contains both files uploaded by a user and key rings containing keys. Each key then knows if the file pointed to in the MediaWiki is another key ring or a file.

4.3 Integration

The overall system design has been covered in section 4.1, but before actually implementing the system, there are a lot of different ways to use the system that have not been clarified.
This section will cover the more technical designs of how to use the program. This is done in a series of sequence diagrams where each sequence diagram will include a brief description on when they are used and any needed assumptions. Then a step by step description of how the actual sequence diagram is done, where each step will include a number that will be used in the description.

4.3.1 Initialize System

Initializing the system is when the user decides to start the application. The user would input the master key, and the program finds the initial key ring in the MediaWiki that will be downloaded, verified and decrypted. The program will then run through the key ring and look up each key in the key ring. Each key points to a file in the MediaWiki and this can be either a key ring or a regular file. Each file found through the key ring is downloaded, verified and decrypted. Verification of each file is done by using each key's read-key. The sequence diagram can be seen in figure 4.2.

Looking at figure 4.2 it can be seen that there is a series of 24 steps before the system has been fully initialized.

1: start is where the user starts the program and includes a configuration file containing a link to a unique key.
2: loadKey and 3 is where the key is loaded using the key_handling library.
4: downloadFile and 5 is where the encrypted key ring is downloaded by

using the key included in the beginning.
6: downloadHash and 7 is where the encrypted hash for the encrypted key ring is downloaded.
8: decryptHash and 9 is where the application uses the cryptography library to verify that the hash was originally written with the correct write-key. This is done by decrypting the encrypted hash with the read-key from the key.
10: verifyFile is where the application generates a hash for the key ring and matches it with the decrypted hash and verifies that they are equal.
11: decryptFile and 12 is where the key ring is decrypted using the symmetric-key and IV from the key included in the beginning.
13: downloadFiles is where the program runs through all keys in the key ring. A key can point to another key ring or a file. Key ring and file are all files in the MediaWiki but are handled differently.
14: downloadFile and 15 is where the encrypted file, from a given key in the key ring, is downloaded.
16: downloadHash and 17 is where the encrypted hash for the encrypted file is downloaded.
18: decryptHash and 19 is where the application uses the cryptography library to verify that the hash was originally written with the correct write-key. This is done by decrypting the encrypted hash with the read-key from the key.
20: verifyFile is where the application generates a hash for the file and matches it with the decrypted hash and verifies that they are equal.
21: decryptFile and 22 is where the file is decrypted. If this is a key ring this will be ran through recursion until all files have been downloaded. If it is a file it is renamed to the name and extension contained in the the key.
23: addFile is where the file gets added to the FUSE file system. This can be done before the mounting happens, so when the folder gets mounted all files previously added will be in the FUSE directory.
24: mount is when all files have been downloaded, verified and decrypted. The rootdir, specified in the configuration file in the beginning, will be mounted and it should include all the downloaded files.

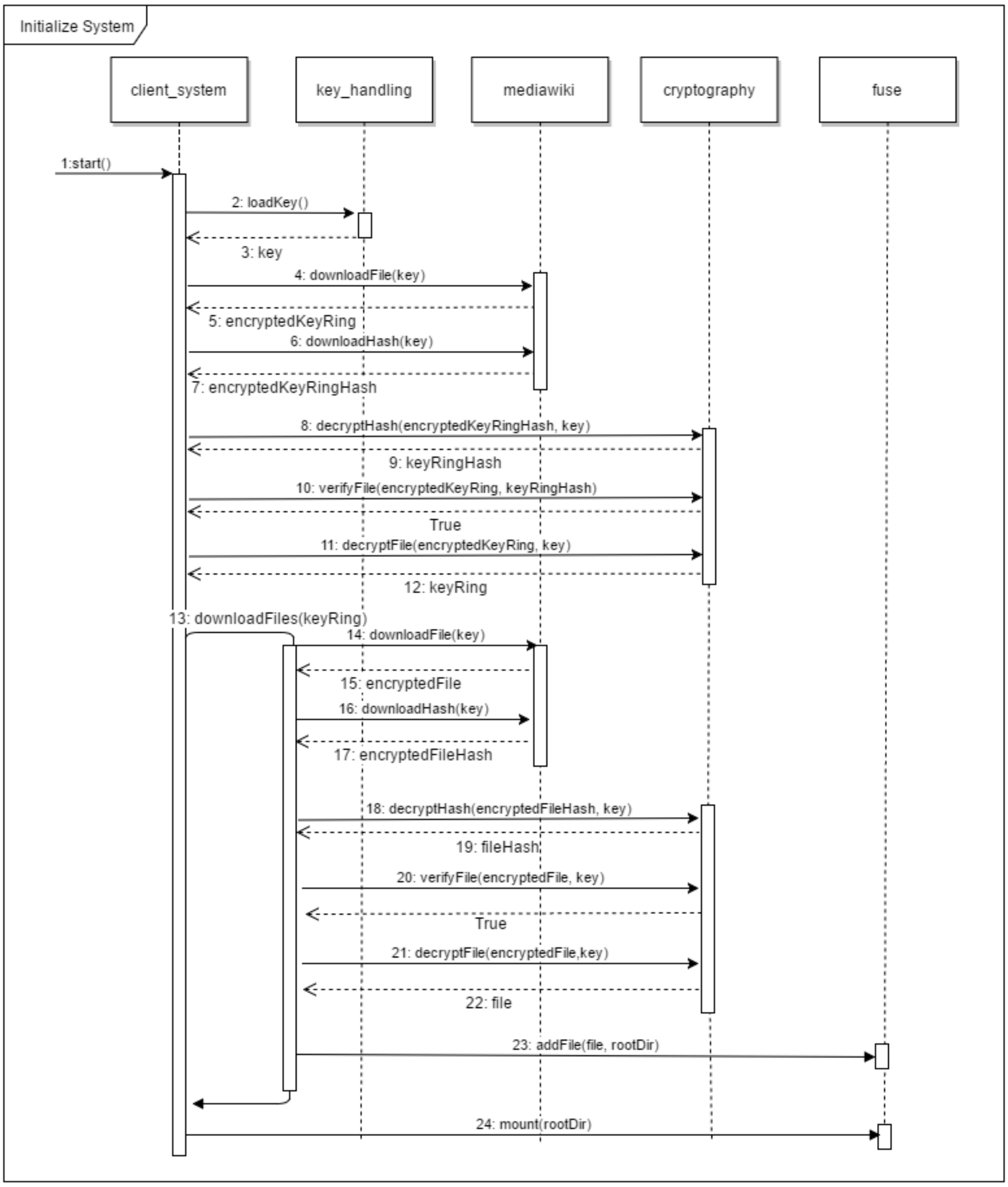


Figure 4.2: Sequence diagram showing the flow for when initializing the system.

4.3.2 Creating File

When the user has initialized the system, the user can work around with files as they want. Essentially the first thing to do would be to create a file. It is assumed that the system has been initialized already before creating a new file. The sequence diagram for when creating a file can be seen at figure 4.3

Looking at figure 4.3 there is a series of 22 steps for when a user creates a file in the FUSE system.

1: createFile and 2 is when the user creates the file, in the mount point or in the file system.
3: createKey and 4 is where the client uses the key_handling library to create a new key for the file. This key includes both read- and write-key.
5: addKey and 6 is where the client adds the key to the key ring using the key_handling library.
7: encryptFile and 8 is where the first created file is encrypted using the created key. This is done using the cryptography library.
9: hashFile and 10 is where the encrypted file gets hashed.
11: encryptHash and 12 is where the hash of the encrypted file gets encrypted with the write-key. This will output an encrypted hash of the file as it will be saved on the MediaWiki.
13: findParentKey and 14 finds the parent key of that key ring. The parent key is the key used to initially encrypt the key ring.
15: encryptFile and 16 is where the key ring is encrypted, using the parent key's symmetric-key and IV.
17: hashFile and 18 is where the encrypted key ring gets hashed.
19: encryptHash and 20 is where the hash of the encrypted key ring gets encrypted with the parent key's write-key. This will output an encrypted hash of the file as it will be saved on the MediaWiki.
21: uploadFile the encrypted file is uploaded to the MediaWiki along side with the encrypted hash for the file.
22: uploadFile the encrypted key ring is uploaded to the MediaWiki along side with the encrypted hash for the key ring.

It can be seen that it needs some steps just to create a file in the file system. This is because the file has to be encrypted and a verification has to be added next to the file giving other users downloading the file the possibility to verify it.

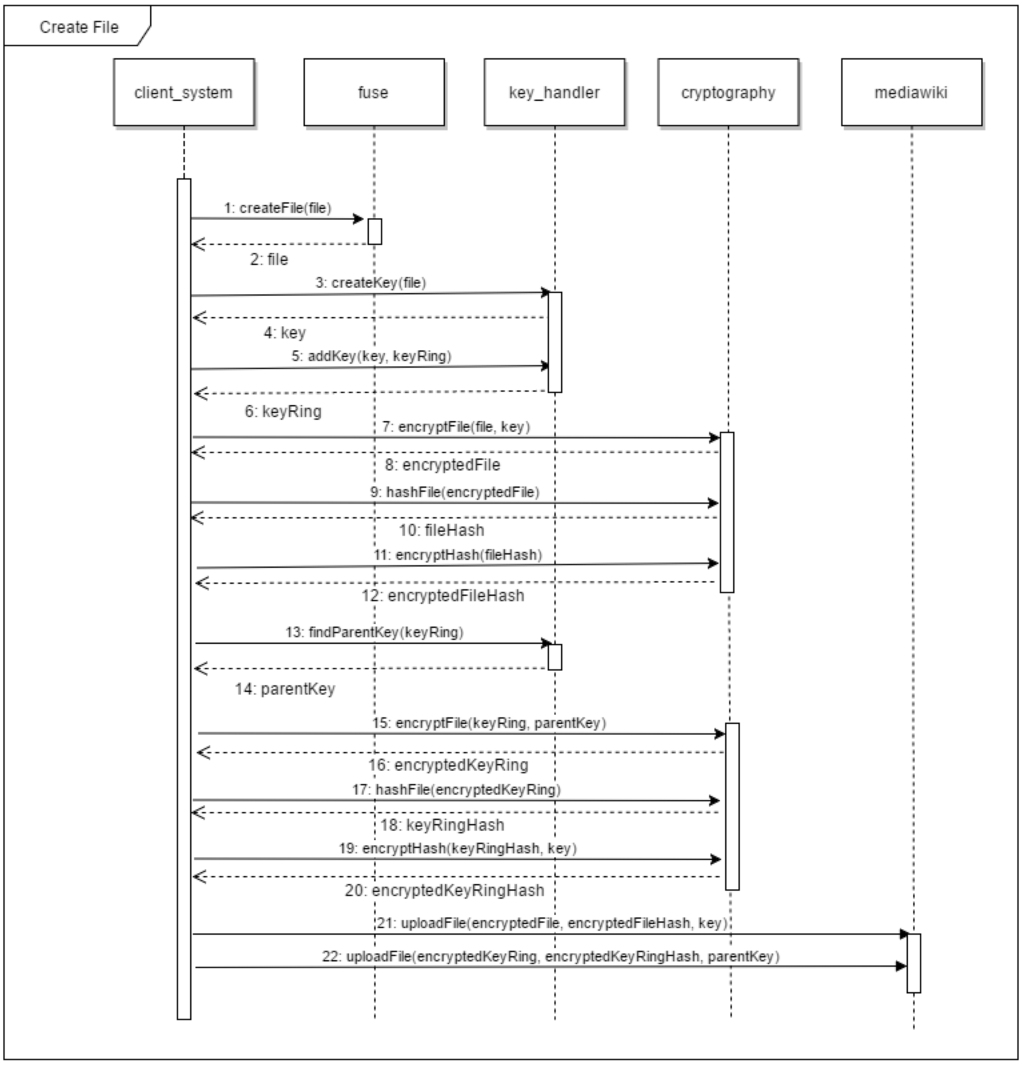

Create File
client_system
fuse
key_handler
cryptography
mediawiki
1: createFile(file)
2: file
3: createKey(file)
4: key
5: addKey(key, keyRing)
6: keyRing
7: encryptFile(file, key)
8: encryptedFile
9: hashFile(encryptedFile)
10: fileHash
11: encryptHash(fileHash)
12: encryptedFileHash
13: findParentKey(keyRing)
14: parentKey
15: encryptFile(keyRing, parentKey)
16: encryptedKeyRing
17: hashFile(encryptedKeyRing)
18: keyRingHash
19: encryptHash(keyRingHash, key)
20: encryptedKeyRingHash
21: uploadFile(encryptedFile, encryptedFileHash, key)
22: uploadFile(encryptedKeyRing, encryptedKeyRingHash, parentKey)

Figure 4.3: Sequence diagram showing the flow for when a user creates a file.

4.3.3 Upload File

While creating a file the system should automatically uploads the file to the MediaWiki. It is assumed that a file has already been created and the system is initialized.
The sequence diagram can be seen at figure 4.4.

The sequence diagram at figure 4.4 shows a series of 9 steps for when a user wants to upload a file already existing.

1: findKey and 2 is where the key_handler finds the key for the file, that the user wants to upload. At this point it should be a part of the key ring.
3: encryptFile and 4 is where the cryptography library is used to encrypt the file, using the found key's symmetric-key and IV.
5: hashFile and 6 is where the encrypted file gets hashed.
7: encryptHash and 8 is where the hash of the encrypted file is encrypted with the write-key. This will output an encrypted hash of the file as it will be saved on the MediaWiki.
9: uploadFile the encrypted file is uploaded to the MediaWiki along side with the encrypted hash.

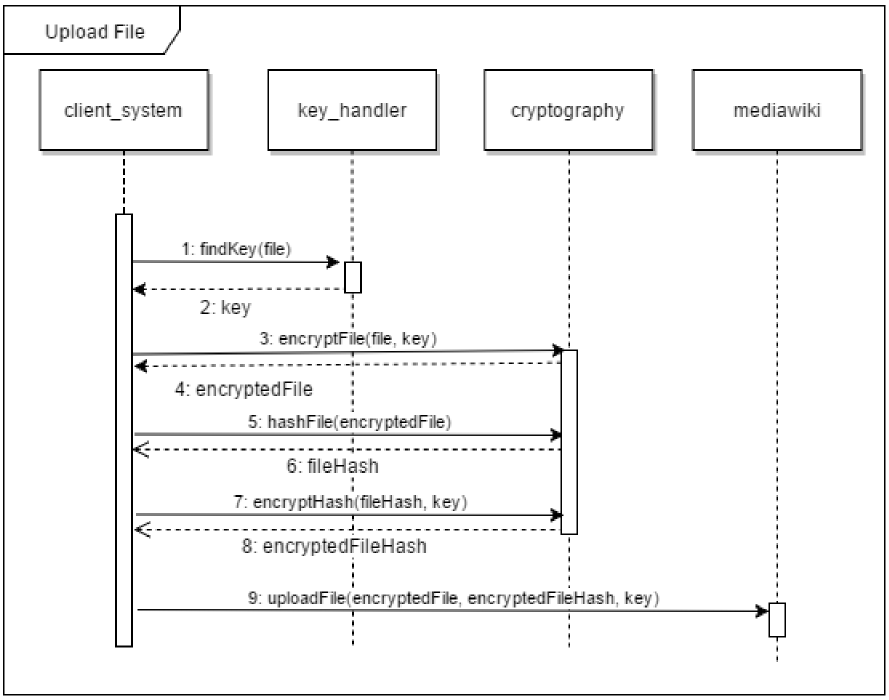

Upload File
client_system
key_handler
cryptography
mediawiki
1: findKey(file)
2: key
3: encryptFile(file, key)
4: encryptedFile
5: hashFile(encryptedFile)
6: fileHash
7: encryptHash(fileHash, key)
8: encryptedFileHash
9: uploadFile(encryptedFile, encryptedFileHash, key)

Figure 4.4: Sequence diagram showing the flow for when a user uploads a file.

4.3.4 Change File

The user often wants to change their files, and as he or she does that the program needs to automatically keep the remote storage synchronized. It is assumed that the system have been initialized and there is a file to change. The sequence diagram of a file change can be seen at figure 4.5

Looking at figure 4.5 it can be seen that when the user changes a file a series of 11 steps is needed.

1: changeFile and 2 is when the user changes the file, and this is automatically detected by the FUSE library.
3: findKey and 4 is where the client uses the key_handler library to find the key for the changed file.
5: encryptFile 6 the file is encrypted by the cryptography library using the symmetric-key and IV from the key found earlier.
7: hashFile and 8 is where the encrypted file gets hashed.
9: encryptHash and 10 is where the hash of the encrypted file gets encrypted with the write-key. This will output an encrypted hash of the file as it will be saved on the MediaWiki.
11: uploadFile the encrypted file is uploaded to the MediaWiki along side with the encrypted hash.

This is all most the same as just uploading a file, but it includes the detecting of a file change from the FUSE library.

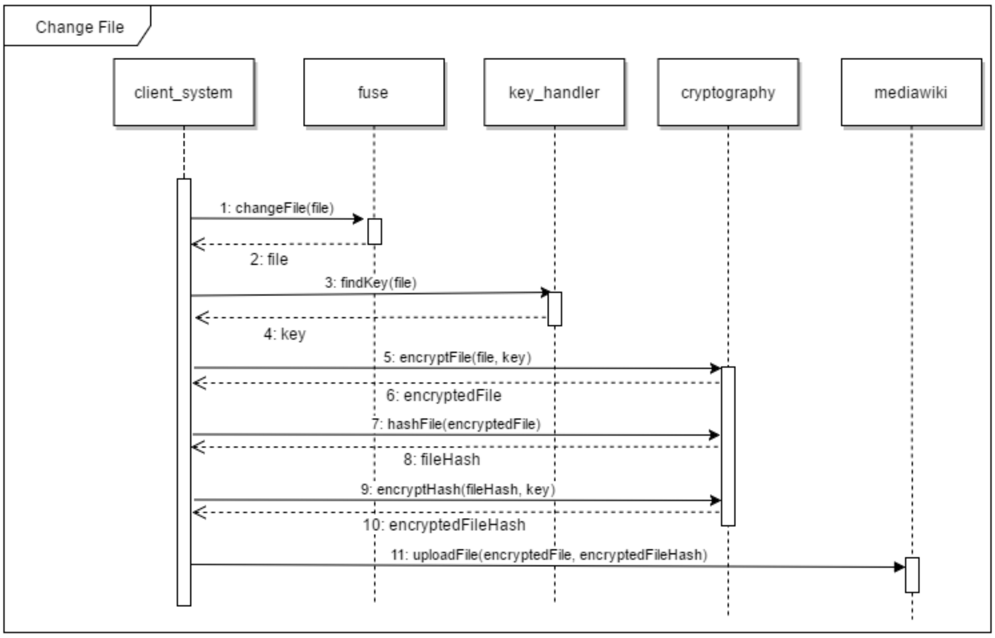


Figure 4.5: Sequence diagram showing the flow for when a user changes a file.

4.3.5 Download File

A lot of the time the user wants to download a file, this could be because another one changed it. It is assumed that the system has been initialized and that the user already has the key to a file stored on the MediaWiki. The sequence diagram for when the user downloads a file is shown at figure 4.6. Looking at figure 4.6 it can be seen that for downloading a file, a series of 10 steps are needed.

1: downloadFile and 2 using the key the encrypted file key gets downloaded from the MediaWiki.
3: downloadHash and 4 is where the encrypted hash for the encrypted file is downloaded.
5: decryptHash and 6 is where the application uses the cryptography library to verify that the hash was originally written with the correct write-key. This is done by decrypting the encrypted hash with the read-key from the key.
7: verifyFile is where the application generates a hash for the file and matches it with the decrypted hash and verifies that they are equal.
8: decryptFile and 9 is where the file gets decrypted using the cryptography library.

10: addFile the file gets added to the FUSE file system, so the user can keep working on it from there.

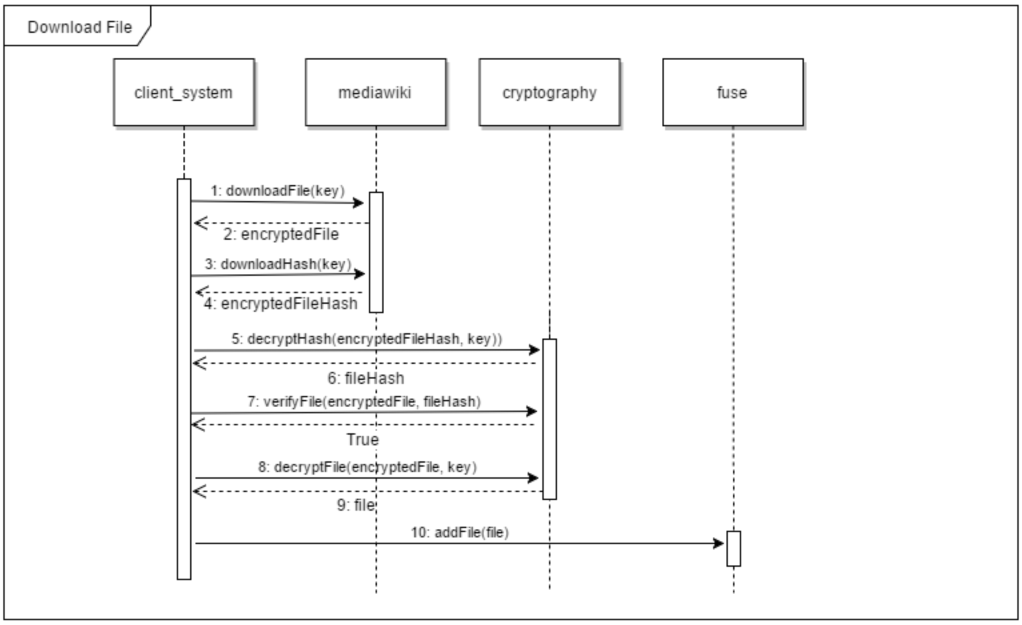


Figure 4.6: Sequence diagram showing the flow for when a user downloads a file.

4.3.6 Download Revision

When a user downloads a file and finds out that it can not be verified because the encrypted file does not match the file's hash on the MediaWiki. It is assumed that the system has been initialized and there is a file that has been uploaded to the MediaWiki multiple times but the last upload is uploaded with a wrong file hash. The sequence diagram of downloading a revision is shown at figure 4.7

Looking at figure 4.7 it can be seen that it requires a series of 19 steps for the system to download a revision.

1: getKey and 2 is where the client uses the key_handling library to get the key for the file.
3: downloadFile and 4 is where the encrypted file gets downloaded from the MediaWiki.
5: downloadHash and 6 is where the encrypted hash for the file gets downloaded.
7: decryptHash and 8 is where the application uses the cryptography library to verify that the hash was originally written with the correct write-key. This is done by decrypting the encrypted hash with the read-key from the key.
9: verifyFile is where the application generates a hash for the file and matches it with the decrypted hash and verifies that they are equal.
10: downloadRevision and 11 is where the encrypted revision is downloaded from the MediaWiki. This is ran because of the verification of the previous file is invalid.
12: downloadRevisionHash and 13 is where the encrypted hash for the file revision is downloaded.
14: decryptHash and 15 is where the application uses the cryptography library to verify that the hash was originally written with the correct write-key. This is done by decrypting the encrypted hash with the read-key from the key.
16: verifyFile is where the application generates a hash for the file and matches it with the decrypted hash and verifies that they are equal.
17: decryptFile and 18 is where the revision file gets decrypted using the cryptography library. This is ran because of the verification of the revision file is valid.
19: addFile is where the file gets added to the mounted folder using the FUSE library. The user can now play around with this revision. When uploading changes to this file, these changes will become the newest version of the file and not be an old revision anymore.

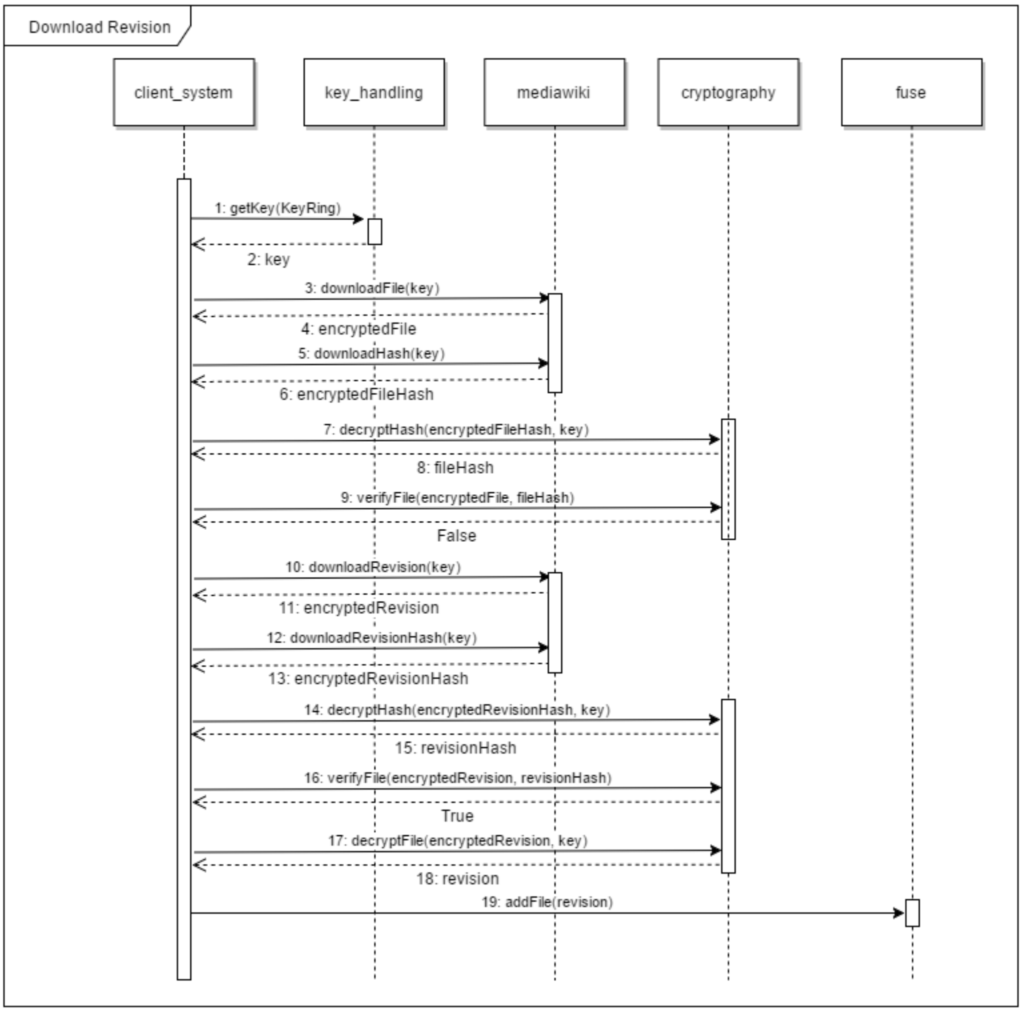

Download Revision
client_system
key_handling
mediawiki
cryptography
fuse
1: getKey(KeyRing)
2: key
3: downloadFile(key)
4: encryptedFile
5: downloadHash(key)
6: encryptedFileHash
7: decryptHash(encryptedFileHash, key)
8: fileHash
9: verifyFile(encryptedFile, fileHash)
False
10: downloadRevision(key)
11: encryptedRevision
12: downloadRevisionHash(key)
13: encryptedRevisionHash
14: decryptHash(encryptedRevisionHash, key)
15: revisionHash
16: verifyFile(encryptedRevision, revisionHash)
True
17: decryptFile(encryptedRevision, key)
18: revision
19: addFile(revision)

Figure 4.7: Sequence diagram showing the flow for when a user downloads a revision of a file.

4.3.7 Rename File

The user will rename files every now and then, and the local file name is decided from the corresponding key. This means that when changing a file's name, the file does not need to be uploaded. Only the change to the key. It is assumed that the system has been initialized and that the user has a key for an uploaded file with write permissions. The sequence diagram for changing a files name is shown at figure 4.8

Looking at the sequence diagram from figure 4.8 it can be seen that a series of 17 steps is needed whenever the user changes a file.

1: renameFile and 2 is where the user actively changes a file in the mounted folder. The FUSE library then detects a file name change.
3: findKey and 4 is where the key is found in the user's key ring, using the key_handler library.
5: changeFileName and 6 is where the file name attribute in the key is changed from the old to the new name.
7: replaceKey and 8 is where the old key is replaced with the new key, in the users key ring. Here the key_handler library is used as well.
9: findParentKey and 10 is where the parent key to the key ring is found using the key_handler library.
11: encryptFile and 12 is where the key ring gets encrypted using the parent key with the cryptography library.
13: hashFile and 14 is where the encrypted file gets hashed.
15: encryptHash and 16 is where the hash of the encrypted file is encrypted with the write-key. This will output an encrypted hash of the file as it will be saved on the MediaWiki.
17: uploadFile the encrypted file is uploaded to the MediaWiki along side with the encrypted hash.

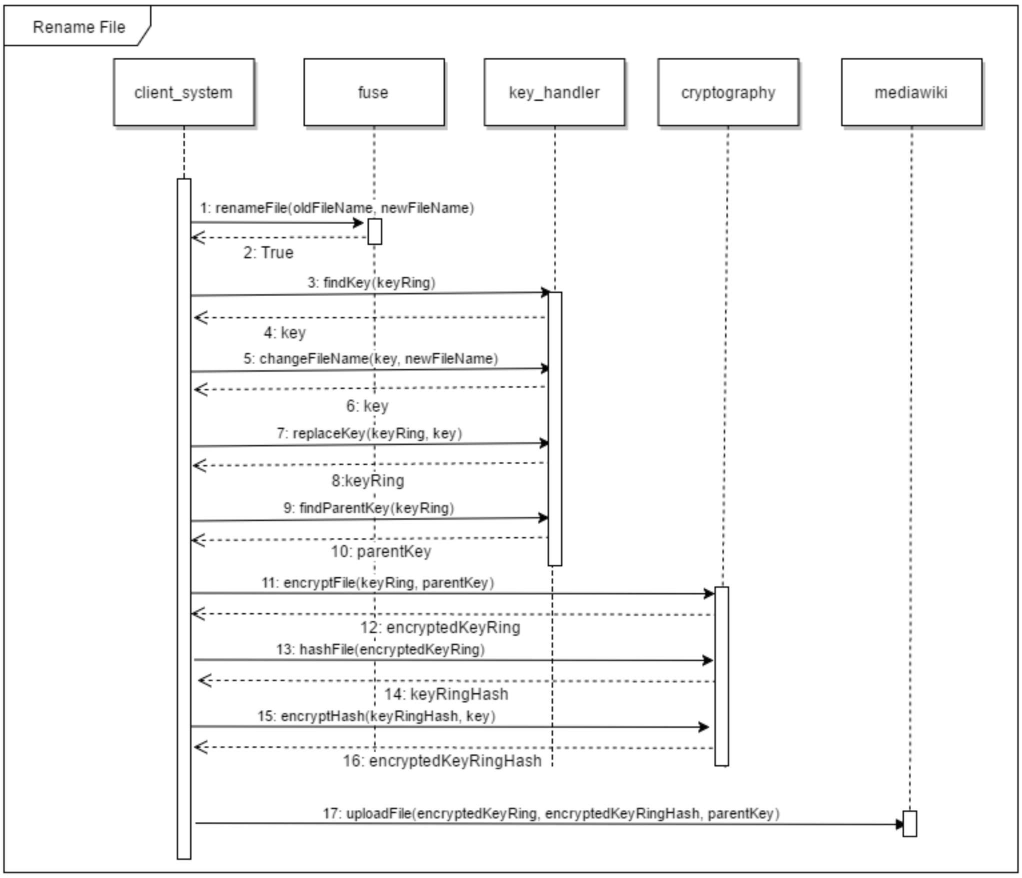

Rename File
client_system
fuse
key_handler
cryptography
mediawiki
1: renameFile(oldFileName, newFileName)
2: True
3: findKey(keyRing)
4: key
5: changeFileName(key, newFileName)
6: key
7: replaceKey(keyRing, key)
8:keyRing
9: findParentKey(keyRing)
10: parentKey
11: encryptFile(keyRing, parentKey)
12: encryptedKeyRing
13: hashFile(encryptedKeyRing)
14: keyRingHash
15: encryptHash(keyRingHash, key)
16: encryptedKeyRingHash
17: uploadFile(encryptedKeyRing, encryptedKeyRingHash, parentKey)

Figure 4.8: Sequence diagram showing the flow for when a user changes a files name.

4.3.8 Delete File

The user might want to delete a file in his or her file system. The file should not be deleted on the MediaWiki, since it should never be possible to do as it would be a major threat to the system. It is assumed that a user already have initialized the system and that a file exists with a key in the system. The sequence diagram shown in figure 4.9 is when a user deletes a file.

Looking at figure 4.9 it can be seen that a series of 16 steps is needed whenever a user deletes a file.

1: deleteFile is where the user deletes a file inside the mounted folder. The FUSE library then detects that a file gets deleted.
2: findKey and 3 is where the key_handler finds the key to the file.
4: findKeyRing and 5 is where the key_handler finds the key ring containing the key.
6: removeKey and 7 is where the found key is removed from the key ring.
8: findParentKey and 9 is where the parent key for the key ring is found. This is needed as the key ring needs to be updated on the MediaWiki as well.
10: encryptFile and 11 is where the key ring gets encrypted with the parent key using the cryptography library.
12: hashFile and 13 is where the encrypted key ring gets hashed.
14: encryptHash and 15 is where the hash of the encrypted key ring is encrypted with the write-key. This will output an encrypted hash of the key ring as it will be saved on the MediaWiki.
16: uploadFile the encrypted key ring is uploaded to the MediaWiki along side with the encrypted hash. Outputting true when succeeding.

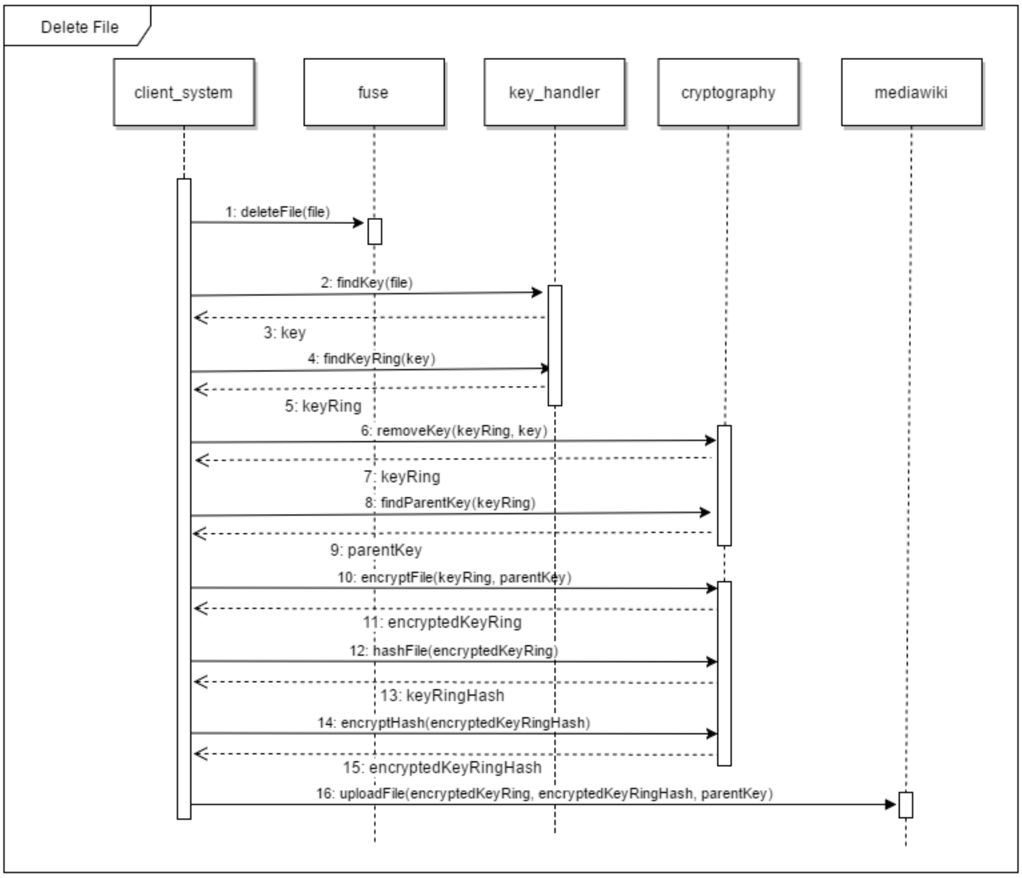

Delete File
client_system
fuse
key_handler
cryptography
mediawiki
1: deleteFile(file)
2: findKey(file)
3: key
4: findKeyRing(key)
5: keyRing
6: removeKey(keyRing, key)
7: keyRing
8: findParentKey(keyRing)
9: parentKey
10: encryptFile(keyRing, parentKey)
11: encryptedKeyRing
12: hashFile(encryptedKeyRing)
13: keyRingHash
14: encryptHash(encryptedKeyRingHash)
15: encryptedKeyRingHash
16: uploadFile(encryptedKeyRing, encryptedKeyRingHash, parentKey)

Figure 4.9: Sequence diagram showing the flow for when a user deletes a file.

4.3.9 Share File

Sharing a file between two users is a little bit difficult, when there is not a place for the users to be created other than locally on their own machine. This means the public-key needs to be shared on a secure line.
It is assumed that the system have been initialized for both user A and B, and that user B has uploaded a key that should be shared with user A. The sequence diagram for sharing a file, can be seen in figure 4.10

Looking at figure 4.10 it can be seen that a series of 35 steps are needed for sharing a key. Some of it includes manually transfer of objects.

1: generateKeys and 2 is where user A generates a key pair consisting of an encryption key (public-key) a decryption key (private-key).
3: sendEncryptionKey is where user A sends the encryption key from the key pair to user A.
4: createSecret and 5 is where user B, creates a one time symmetric-key and IV needed for encrypting the key to be shared with user B.
6: encryptFile and 7 is where user B encrypts the key using the before generated one time symmetric-key and IV. User B can select if there should be included read- and/or write-key.
8: encryptSecret and 9 is where user B, encrypts the symmetric-key and IV using the public-key received from user A.
10: sendKey is where user B, sends the encrypted file and encrypted symmetric-key and IV to user A.
11: decryptSecret and 12 is where user A decrypts the symmetric-key and IV, using the private-key generated in the beginning.
13: decryptFile and 14 is where user A decrypts the key using the symmetric-key and IV decrypted in previous step.
15: downloadFile and 16 user A downloads the encrypted file from the MediaWiki.
17: downloadHash and 18 is where the encrypted hash for the file is downloaded.
19: decryptHash and 20 is where the application uses the cryptography library to verify that the hash was originally written with the correct write-key. This is done by decrypting the encrypted hash with the read-key from the key.
21: verifyFile is where the application uses the cryptography library to verify that the encrypted hash was originally written with the correct write-key. This is done by decrypting the encrypted hash with the read-key from the key and matching it with the file just downloaded.
22: decryptFile and 23 is where user A decrypts the file using the symmetric-key and IV from the key decrypted earlier.
24: addFile is where the file is added to the mounted directory using the FUSE library. Changes made by user A can be uploaded the MediaWiki, if user B included a write-key in the key sent to user A.
25: addKey 26 is where user A adds the key to his own key ring.
27: findParentKey and 28 is where user A finds the parent key to the key ring.
29: encryptFile and 30 is where user A encrypts the key ring using the parent key found in previous step.
31: hashFile and 32 is where the encrypted key ring gets hashed.
33: encryptHash and 34 is where the hash of the encrypted key ring is encrypted with the write-key. This will output an encrypted hash of the file as it will be saved on the MediaWiki.
35: uploadFile the encrypted key ring is uploaded to the MediaWiki along side with the encrypted hash.

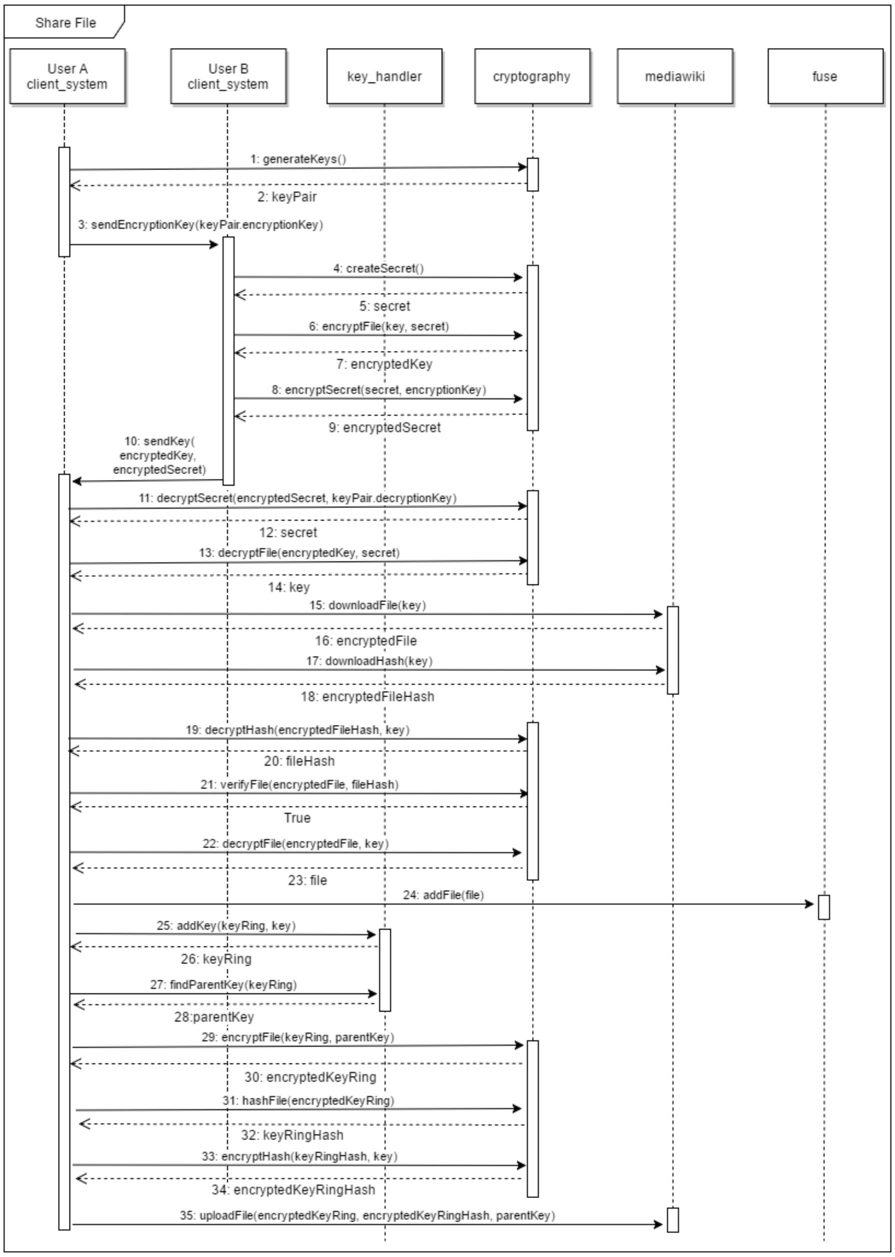

Share File
User A client_system
User B client_system
key_handler
cryptography
mediawiki
fuse
1: generateKeys()
2: keyPair
3: sendEncryptionKey(keyPair.encryptionKey)
4: createSecret()
5: secret
6: encryptFile(key, secret)
7: encryptedKey
8: encryptSecret(secret, encryptionKey)
9: encryptedSecret
10: sendKey(encryptedKey, encryptedSecret)
11: decryptSecret(encryptedSecret, keyPair.decryptionKey)
12: secret
13: decryptFile(encryptedKey, secret)
14: key
15: downloadFile(key)
16: encryptedFile
17: downloadHash(key)
18: encryptedFileHash
19: decryptHash(encryptedFileHash, key)
20: fileHash
21: verifyFile(encryptedFile, fileHash)
True
22: decryptFile(encryptedFile, key)
23: file
24: addFile(file)
25: addKey(keyRing, key)
26: keyRing
27: findParentKey(keyRing)
28:parentKey
29: encryptFile(keyRing, parentKey)
30: encryptedKeyRing
31: hashFile(encryptedKeyRing)
32: keyRingHash
33: encryptHash(keyRingHash, key)
34: encryptedKeyRingHash
35: uploadFile(encryptedKeyRing, encryptedKeyRingHash, parentKey)

Figure 4.10: Sequence diagram showing the flow for when a user wants to share a file with another user.

4.3.10 Remove Sharing

Whenever a user removes a sharing to another user, unfortunately he or she cannot remove the sharing to the current file. What the user can do is simply to create a new key for the changed file and upload the file as a new file, where the user does not give access to the user who needed to be removed from the sharing.
The reason that it is not possible to remove a sharing for an existing key is that it would be possible for all users having access to the key be able to remove users from it. The only access control added to sharing is read- and/or write permissions.
It is assumed that the system has been initialized and that there is a file where the user have this files key, and that it is shared with another user, but the key ring is not shared. The sequence diagram for removing a sharing can be seen at figure 4.11.

Looking at figure 4.11 it can be seen that a series of 13 steps is needed whenever a user needs to re-create a key and in that way "remove" a sharing.

1: createKey and 2 is where the key_handler library is used to create a new key for the file.
3: addKey and 4 is where the key_handler library is used to add the newly created key into the key ring.
5: findParentKey and 6 finds the parent key to the key ring as it is needed to upload the current changes to the key ring to the MediaWiki.
7: encryptFile and 8 is where the key ring is encrypted using the cryptography library.
9: hashFile and 10 is where the encrypted key ring gets hashed.
11: encryptHash and 12 is where the hash of the encrypted key ring is encrypted with the write-key. This will output an encrypted hash of the file as it will be saved on the MediaWiki.
13: uploadFile the encrypted key ring is uploaded to the MediaWiki along side with the encrypted hash.

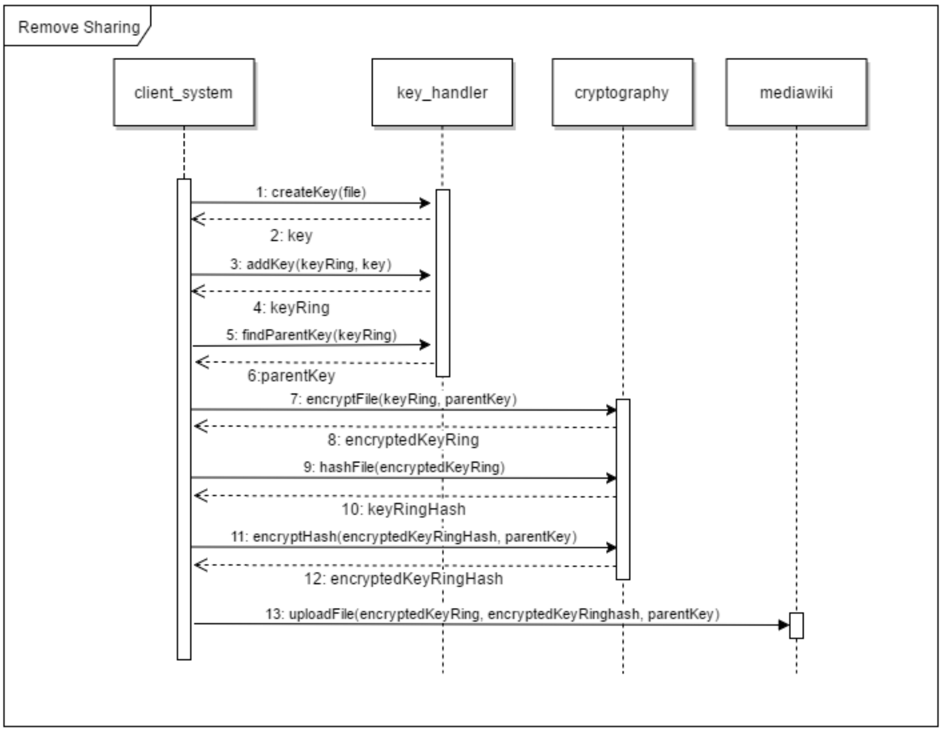

Remove Sharing
client_system
key_handler
cryptography
mediawiki
1: createKey(file)
2: key
3: addKey(keyRing, key)
4: keyRing
5: findParentKey(keyRing)
6:parentKey
7: encryptFile(keyRing, parentKey)
8: encryptedKeyRing
9: hashFile(encryptedKeyRing)
10: keyRingHash
11: encryptHash(encryptedKeyRingHash, parentKey)
12: encryptedKeyRingHash
13: uploadFile(encryptedKeyRing, encryptedKeyRinghash, parentKey)

Figure 4.11: Sequence diagram showing the flow for when a user removes a sharing to a file.

4.3.11 End Session

Sooner or later the user is done using the application and when that happens all files needs to be uploaded to the MediaWiki. They could then be either stored encrypted locally on the computer or simply deleted. In this project they will be deleted as when starting the project, the user needs to download the file when initializing.
It is assumed that the system has been initialized and that the user has all the keys and files in the current session. The sequence diagram for ending the session is shown in figure 4.12

Looking at figure 4.12 it can be seen that a series of 20 steps are needed when the user closes the application.

1: end is where the user tells the application that it should close down.
2: getKeyRing and 3 is where the user's key ring is loaded that was retrieved

using the key in the configuration in the initialization.
4: uploadFiles is where the system runs through all keys in the key ring and do the operations needed on them.
5: encryptFile and 6 is where the system encrypts the given file using the given key from the key ring.
7: hashFile and 8 is where the encrypted file gets hashed.
9: encryptHash and 10 is where the hash of the encrypted file is encrypted with the write-key. This will output the content as it will be saved on the MediaWiki as the file content.
11: uploadFile is where the encrypted file is uploaded to the MediaWiki along side with the hash of the file.
12: removeFile is where the FUSE library removes the file from the mounted folder.
13: encryptFile and 14 is where the key ring is encrypted using the master key with the cryptography library.
15: hashFile and 16 is where the encrypted key ring gets hashed.
17: encryptHash and 18 is where the hash of the encrypted key ring is encrypted with the parent keys write-key. This will output an encrypted hash of the key ring as it will be saved on the MediaWiki.
19: uploadFile the encrypted key ring is uploaded to the MediaWiki along side with the hash.
20: unmount is where the mounted folder gets unmounted using the FUSE library.

All files have now been uploaded to the MediaWiki so the application is ready to be closed.

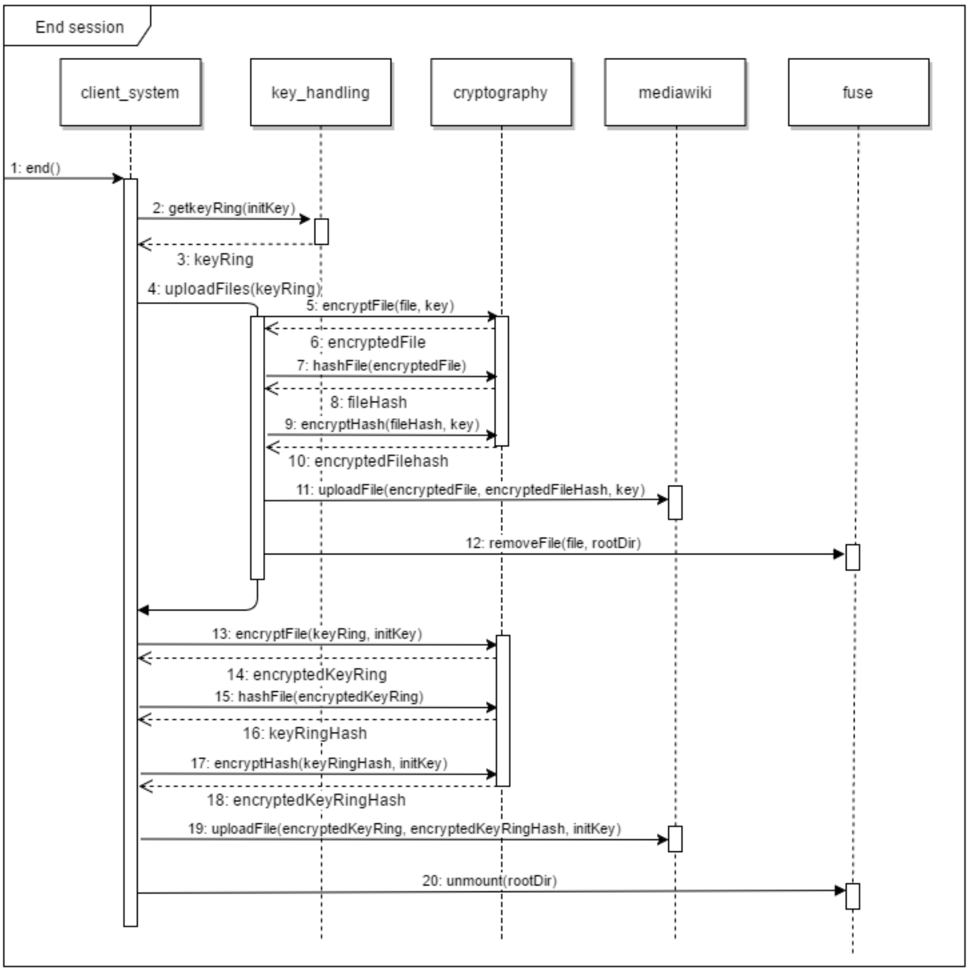

End session
client_system
key_handling
cryptography
mediawiki
fuse
1: end()
2: getkeyRing(initKey)
3: keyRing
4: uploadFiles(keyRing)
5: encryptFile(file, key)
6: encryptedFile
7: hashFile(encryptedFile)
8: fileHash
9: encryptHash(fileHash, key)
10: encryptedFilehash
11: uploadFile(encryptedFile, encryptedFileHash, key)
12: removeFile(file, rootDir)
13: encryptFile(keyRing, initKey)
14: encryptedKeyRing
15: hashFile(encryptedKeyRing)
16: keyRingHash
17: encryptHash(keyRingHash, initKey)
18: encryptedKeyRingHash
19: uploadFile(encryptedKeyRing, encryptedKeyRingHash, initKey)
20: unmount(rootDir)

Figure 4.12: Sequence diagram showing the flow for when a user closes the file system.

4.4 Cryptography Design

The functionalities have now been designed using sequence diagrams, and they show the flow of the application. However, they do not tell anything about which cryptography algorithms that should be used. It is needed to choose a symmetric algorithm for file encryption / decryption and a asymmetric for both key encryption and for verification of the file. Hashing is a part of the verification as well.
All methods and the choices for algorithms are listed in table 4.1, and afterwards described why they have been chosen.

Operation	Algorithm	Mode of Operation	Key length
File encryption	AES	CBC	256
File decryption	AES	CBC	256
Key sharing	RSA	None	2048
File verification	RSA	None	2048
File hashing	SHA256	None	None

Table 4.1: The different chosen cryptography algorithms needed.

In 2003 RSA Security claimed that a 1024-bit RSA key is equivalent to a 80-bit symmetric-key. 2048-bit RSA key is equivalent to a 112-bit symmetric-key and 3072-bit RSA key is equivalent to a 128-bit symmetric-key.
It is also claimed that a 1024-bit RSA key is likely to become crackable some time between 2006 and 2010. 2048-bit RSA key should be sufficient until 2030 [NISc] and that is why the key length of RSA in this project decided to be 2048.

4.4.1 File Encryption and Decryption

When encrypting and decrypting files performance can very quickly become an issue as files in general growing rapidly. This is why it is a must to use symmetric encryption as the performance is way better than when using asymmetric encryption.
There is used AES CBC 256-bit as this is a NIST Standard [NISb] and this is considered a very good algorithm for the purpose. If it come to the matter of performance AES can also be implemented to be very lightweight without compromising the security level.
Performance is not a part of the scope in this project, so other algorithms could be used as well, however as the security is not worse when using AES it is still chosen.

4.4.2 File Hashing and Verification

All files are encrypted with a symmetric-key at first, then after this they are hashed using SHA256. When the files have been encrypted and hashed, the hash is encrypted using a private-key, called the write-key. For hashing SHA256 is chosen as this is minimum recommendation from NIST [nisa].
MD5 hashing could be considered as it is not password hashing, but since NIST recommends not to use it at all.
It is mentioned that a write-key is needed to encrypt the file hash afterwards, and this requires an asymmetric algorithm to do so. Then the public-key can be considered the read-key as this can be used to verify the right write-key has been used to encrypt. As the performance is not a problem when the files have been hashed first (these hashes have a fixed output size), RSA can be used for this as well.

4.5 Summary

The design of the application have now been specified. This was done by first making an overall design on which different libraries needed and where these should be included. Shortly how the key handling is done, and next a lot of the given use cases was created as sequence diagrams. In the end the cryptography design was chosen. All this was done to get ready for the implementation.

Chapter 5

Implementation

The implementation chapter will be more in depth on how the system actually have been implemented. In the theory chapter 2 the tools where researched, in the requirements chapter 3 a more specific list of requirements was setup for both cryptography and the technical point of view. Furthermore, the design chapter 4 handled how the system is designed.
However, as in all projects the implementations and design is not always the same and the implementation will specify more directly how different methods are implemented. Firstly an overview of the implementation is shown, then how the cryptography choices was implemented and which libraries was chosen for this. The key- and data structure is looked at and in the end there will be a description for different methods implemented.

5.1 Overview of Implementation

Looking at the overview of the system there is a lot of different libraries created for this specific system. The overall implementation is shown at figure 5.1 and it can be seen that the implementation is splitted into three different subjects.

- Client System

- Libraries
- Web server

Looking at figure 5.1 it can be seen that the user uses the system through the client system. In the client system there is the client it self and the file system, and the client is basically a console application where the user can give commands to the system and the file system is all the file operations.
Next to see at is the Libraries involved in the system and looking at 5.1 it can be seen there is six different libraries included in this section. Cryptography is basically all the cryptography operations, including symmetric- and asymmetric encryption / decryption as well as hashing. Next is the Key library which includes all methods regarding keys. MediaWiki is the library for handling the communication with the MediaWiki, it is basically an imported library with some changes.
The user needs to specify the config (see section 5.8.1). The JAXBContext operations handles the XML parsing and the XmlValidator validates the given XML by comparing it with the associated XML schema for the XML.
The web server is the standard MediaWiki setup with a few changes to the LocalSettings.php file. It is running on an apache2 server, with MySQL database. No major changes to the default setups.

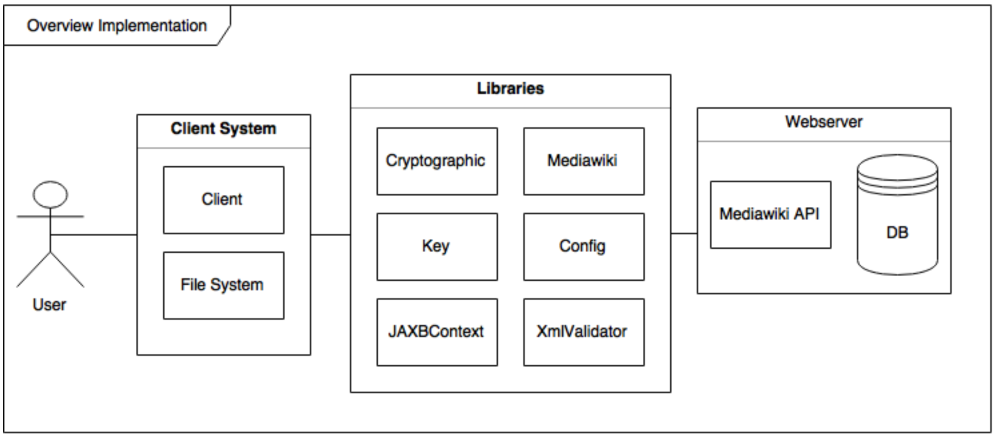


Figure 5.1: The overall view of the implementation

The programming language for the implementation is Oracle's Java version eight [Ora], as there is a lot of good standard libraries for both encryption / decryption, hashing, file handling and more. Furthermore, there exists some decent open libraries that includes the MediaWiki operations.

5.2 Encoding

When shipping binary data across a network it is important to encode the data to prevent interpret with other protocols by eliminating special characters. The encoder used in this project is Base64 because it rely on the same 64 characters being present in many character sets.
When creating the encode function in java the library java.util.Base64 is used. This library is able to both encode and decode data by having static functions for a Base64 encoder and decoder.
When encoding data the Base64 library has a static function to get the encoder using the method getEncoder(). This encoder has a function for encoding an array of bytes and returning the encoded data as a string.
When decoding a Base64 string the Base64 library has a static function to get the decoder using the method getDecoder(). This decoder has a function for decoding a string and returning an array of bytes.

Listing 5.1: Encoding and deconding using Base64

```
Base64.getEncoder().encodeToString(bytes); // Encoding

Base64.getDecoder().decode(encoding);      // Decoding
```

5.3 File stream

Accessing files across a network is done by using a file stream. A file stream is used when retrieving data from a source or writing data to a destination. In Java there are two libraries for reading a file stream and writing to a file stream, java.io.InputStream and java.io.OutputStream. These libraries are abstract classes and is the superclass of all other classes representing InputStream and OutputStream.
In this project the only subclass used is FileInputStream and FileOutputStream.

Using the Java library java.io.FileInputStream it is possible to initialize a FileInputStream. A FileInputStream is only used to read data from a source. The initialization of the FileInputStream takes either a file or a string file path as input. When having the FileInputStream initialized it is possible to read data from the source, given in the initialization, by using the function read(byte b[]). The read function stores all the bytes read from the stream into the buffer array b. Once the FileInputStream is read it does not contain the data and if the data is needed to be read once again, the FileInputStream needs to be reinitialized.

When writing a file the Java library java.io.FileOutputStream is used to initialize a FileOutputStream. A FileOutputStream is only used to write data to a destination. The initialization of FileOutputStream takes the same inputs as FileInputStream. When having the FileOutputStream initialized it is possible to write data to a destination, given in the initialization, by using the function write(byte b[]). However, this function is rarely used because normally the only writing done is copying bytes from a source to a destination. Copying data from a source to a destination can be done with the Java library org.apache.commons.io.IOUtils that contains a copy function taking an InputStream and OutputStream as input. This function simply just copies the data in the InputStream to the OutputStream. This allows streams up to 2GB to be copied. When copying data streams greater than 2GB another copy function is used in the library.

When using InputStream and OutputStream it is important to make sure to close the stream after it has been used to release any system resources associated with the stream.

5.4 Unique identifier

Storing and retrieving files in the cloud needs to be done with a unique identifier for each file. The identifier is needed when the file is later retrieved or changed by a user. Using Globally Unique Identifier also called GUID allows to create a unique reference number. GUIDs are stored as 128-bit values and are displayed as 32 hexadecimal digits with groups seperated by dashes (-).
The total number of unique GUIDs is 2^{122} and gives the probability of having a duplicate as negligible.
By using the Java library java.util.UUID it is possible to generate a random GUID. Every file stored in the cloud are stored with random generated GUID.

5.5 Cryptography Library

Cryptography is an important aspect in this project to make sure the files stored in the cloud is only accessible by the owners them self. One of the cryptography methods that is needed is a hash function for generating a checksum of the files. This is needed to check if two files are the same by comparing both files checksum. Another cryptography method is symmetric cryptography and is used when encrypting and decrypting files. Symmetric cryptography is used for encrypting files because it is fast and efficient. The last cryptography method that is needed is the asymmetric cryptography. Among other things asymmetric cryptography is used to generate a digital signature to verify read and write permissions.

5.5.1 Hashing

Hashing is used to generate a checksum for a file. This is used to verify that the file downloaded and uploaded are the same by comparing the file downloaded with its checksum.
When creating the hash function the Java library java.security.MessageDigest is used. Message digests are one-way hash functions that take arbitrary-sized data and output a fixed-length hash value. The library provides several hash functions such as SHA-1, MD5, SHA-256, etc.
By initializing a MessageDigest with the hash function SHA-256, the MessageDigest can now be updated with bytes. The update function updates the digest using the specified byte array and an offset and limit of bytes to use.

Listing 5.2: Generating checksum for a file

```
MessageDigest md;
md = MessageDigest.getInstance(_DIGEST_ALGORITHM);

byte[] buffer = new byte[_READ_BYTE_LENGTH];
int len;
while ((len = stream.read(buffer)) >= 0)
{
    md.update(buffer, 0, len);
}
```

5.5.2 Symmetric cryptography

Symmetric cryptography is used for encrypting files uploaded to the cloud. The algorithm used is AES-256 with CBC and PKCS5Padding. This requires a secret and an initialization vector when encrypting and decrypting data.
The Java library javax.crypto.Cipher has been used to encrypt and decrypt data. The initialization of the cipher is done by using the static function getInstance(String var0) where the input parameter is the algorithm that is going to be used for the cipher, in this case AES/CBC/PKCS5Padding. When having the cipher initialized it can then be set to either encrypt or decrypt data. This is

done by using the function init(int var1, Key var2, AlgorithmParameterSpec var3) on the initialized cipher. Where var1 sets if it should encrypt or decrypt data, var2 is the secret and var3 is the initialization vector. Now the cipher is all set and ready for either encrypt or decrypt data with the specified secret and initialization vector.
This cipher is only used to encrypt and decrypt files in the project and to do that another Java library javax.crypto.CipherOutputStream is needed. The CipherOutputStream is initialized by giving it an OutputStream (described earlier in section 5.3) and the cipher just created. Now when writing the output stream the cipher will make sure it gets encrypted.

Listing 5.3: Encrypting a file using Cipher

```
public void encrypt(InputStream in, OutputStream out)
    Cipher cipher = Cipher.getInstance(this.
        _PADDING_SCHEME);
    // secretKey and IV known from the initialization of
        the object
    cipher.init(Cipher.ENCRYPT_MODE, this._secretKey,
        this._iv);

    out = new CipherOutputStream(out, cipher);
    IOUtils.copy(in, out);

    in.close();
    out.close();
}
```

Listing 5.4: Decrypting a file using Cipher

```
public void decrypt(InputStream in, OutputStream out) {
    Cipher cipher = Cipher.getInstance(this.
        _PADDING_SCHEME);
    // secretKey and IV known from the initialization of
        the object
    cipher.init(Cipher.DECRYPT_MODE, this._secretKey,
        this._iv);

    in = new CipherInputStream(in, cipher);
    IOUtils.copy(in, out);

    in.close();
    out.close();
}
```

5.5.3 Asymmetric cryptography

Asymmetric cryptography is used to grant users read- and/or write permissions and key sharing. The algorithm used is RSA 2048-bit. The RSA requires a KeyPair consisting of a public- and a private-key. KeyPair is generated by using the Java library java.security.KeyPair and java.security.KeyPairGenerator. The KeyPairGenerator is initialized by using the static function KeyPairGenerator.getInstance(String algorithm) where the parameter specifies what cryptography algorithm is going to be used, in this case RSA. Then the KeyPairGenerator object needs to know the key length which is done by using the function initialize(int keysize), in this case 2048. Now a KeyPair, containing the public- and private-key, can be extracted from the KeyPairGenerator by using the function generateKeyPair().

Listing 5.5: Generating KeyPair

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance(this.
    _CRYPTOGRAPHY_ALGORITHM);
kpg.initialize(this._KEY_LENGTH);
KeyPair kp = kpg.generateKeyPair();
kp.getPublic().getEncoded();
kp.getPrivate().getEncoded();
```

When the KeyPair is generated it is possible to encrypt and decrypt using either the public- or private-key. That means there are following four functions that need to be declared:

- Encrypting using the private-key
- Encrypting using the public-key
- Decrypting using the private-key
- Decrypting using the public-key

In all functions a cipher needs to be declared by using the Java library javax.crypto.Cipher. The cipher is initialized by using the static function getInstance(String var0) where the parameter is the algorithm used, in this case RSA. When the cipher is initialized it needs to be set if it should encrypt or decrypt and the secret used for the operation. This is done by using the function init(int var1, Key var2) where var1 decides if it should encrypt or decrypt and var2 is the public- or private-key. When the mode of operation is set the cipher function doFinal(byte[] var1) can be used where var1 is the plain text or cipher text that needs to be encrypted or decrypted.

Listing 5.6: Asymmetric cryptography encryption using private-key

```
public void encryptByUsingprivate-key(String plainText) {
    cipher = Cipher.getInstance(this._CIPHER_ENCRYPTION);
    // private-key known from the initialization of the
        object
    cipher.init(Cipher.ENCRYPT_MODE, this._getprivate-key
        ());
    cipher.doFinal(plainText.getBytes("UTF-8"));
}
```

Listing 5.7: Asymmetric cryptography encryption using public-key

```
public void encryptByUsingpublic-key(String plainText) {
    cipher = Cipher.getInstance(this._CIPHER_ENCRYPTION);
    // public-key known from the initialization of the
        object
    cipher.init(Cipher.ENCRYPT_MODE, this._getpublic-key
        ());
    cipher.doFinal(plainText.getBytes("UTF-8"));
}
```

Listing 5.8: Asymmetric cryptography decryption using private-key

```
public void decryptByUsingprivate-key(String cipherText)
   {
    cipher = Cipher.getInstance(this._CIPHER_ENCRYPTION);
    // private-key known from the initialization of the
        object
    cipher.init(Cipher.DECRYPT_MODE, this._getprivate-key
        ());
    cipher.doFinal(Encoding.decode(cipherText));
}
```

Listing 5.9: Asymmetric cryptography decryption using public-key

```
public void decryptByUsingpublic-key(String cipherText) {
    cipher = Cipher.getInstance(this._CIPHER_ENCRYPTION);
    // public-key known from the initialization of the
        object
    cipher.init(Cipher.DECRYPT_MODE, this._getpublic-key
        ());
    cipher.doFinal(Encoding.decode(cipherText));
}
```

The private-key (write-key) is used to sign the file when uploading it to the cloud. It starts by generating a checksum for the encrypted file and encrypting the checksum with the private-key and adds it next to the file. Now when a user downloads the file and have the public-key (read-key), it is possible for the user to verify that the file is legit by decrypting the checksum and comparing it with a checksum generated for the downloaded encrypted file. If they match the user can be certain that the file is legit and that the user who uploaded the file actually had write permissions. If the checksum does not match or simply the checksum is not defined, the user knows that there is a mismatch and that the file is not legit. The user now tries an older revision and continues to do this until there is a match.

5.6 Key Structure

This section includes the structure of the keys and how it will be shown in the XML format as they are used in the implementation. There are two kind of key types in the implementation, FileKey and KeyRing. This is given by the type attribute in the key structure and is used to know if the file pointing to (also called the reference) is a KeyRing containing other keys or just a normal file.
Typically the file will not contain a single key, but be a KeyRing. However, looking first at the structure of one key, and then afterwards for a KeyRing. Looking at listing 5.10 the structure of one key can be seen. Firstly the id of the key is an XML attribute and the same goes for type where as the rest of the objects are XML elements.
Some of the XML elements are the symmetric-key and IV (initialization vector) that are used to encrypt and decrypt the reference.
The read-key and write-key are the public-key and private-key. These are used to allow read- and/or write permissions.
The reference is the id of the file that has been decrypted with the key.
The MediaWiki element contains url, file-extension, and optional username and password. Each key contains a MediaWiki element to make it easier to share files across different MediaWikis.
Finally there is a date for when the key has been added.

Listing 5.10: Example of Key structure in XML.

```
<key id="397b5be3-d0f9-40a8-b47c-9e696db6081c" type="
    KeyRing">
    <symmetric-key>
        lWPgsgCcdbWN1X1nHe6ztqfSDMPrq3WO1j4lCAwzvGk=</
        symmetric-key>
    <IV>LBnmODwmSG/usGhkqGZA+Q==</IV>
    <read-key>MIIBI...</read-key>
    <write-key>MIIEv...</write-key>
    <reference>ae6be7bc-5a3e-4a8f-9e4b-f570eba66fcb</
        reference>
    <MediaWiki>
        <url>https://localhost/MediaWiki</url>
        <username>user1</username>
        <password>user1</password>
        <file-extension>.box</file-extension>
    </MediaWiki>
    <added>2016-05-10T20:22:44</added>
</key>
```

Looking at the type it can be seen that this specific example is linking to a key ring. The needed information in a key is the symmetric-key with the initialization vector (used in CBC mode in AES). Next is the read- and the write-key used for RSA. The reference object, is the id of the object this key is referring to, in listing 5.10 it is empty, but this would typically be a string with the id. The name object is the name of the object, this is only used when it is a file key as it will be the name of the file, when it is downloaded.
Next is the information for the MediaWiki used for the object this key is linking too. One user can use more than one MediaWiki, so this information is needed on each key as it might be different. The last is when the key was added.
The next object is the key ring and this can be seen in listing 5.11, where this also have an id as the keys does.

Listing 5.11: Example of KeyRing structure in XML.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<key-ring id="ae6be7bc-5a3e-4a8f-9e4b-f570eba66fcb">
    <key id="41b7d1d8-9b51-4718-9132-acda8dee4076" type="
        FileKey">
     ..
    </key>
     ..
    <key id="b0da69bd-885e-4c33-bb97-4c6b8d65ba8a" type="
        KeyRing">
     ..
    </key>
    <last-updated>2016-05-10T20:22:44</last-updated>
</key-ring>
```

The id is an XML Attribute where the rest are XML elements. The key ring is basically an id, list of keys with XML format and then when it was last updated. When working with XML files, they need to have a corresponding XML schema, as they are defining the structure of a given XML File.

Listing 5.12: Key structure in XSD (XML Schema) format.

```
<xs:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified" xmlns:xs="http://www.w3
    .org/2001/XMLSchema">
    <xs:complexType name="MediaWikiType">
        <xs:sequence>
            <xs:element type="xs:string" name="url"/>
            <xs:element type="xs:string" minOccurs="0"
                maxOccurs="1" name="username"/>
            <xs:element type="xs:string" minOccurs="0"
                maxOccurs="1" name="password"/>
            <xs:element type="xs:string" name="file-
                extension"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="KeyType">
        <xs:sequence>
            <xs:element type="xs:string" name="symmetric-
                key"/>
            <xs:element type="xs:string" name="IV"/>
            <xs:element type="xs:string" name="name"
                minOccurs="0" maxOccurs="1" />
            <xs:element type="xs:string" name="read-key"
                minOccurs="0" maxOccurs="1" />
            <xs:element type="xs:string" name="write-key"
                 minOccurs="0" maxOccurs="1" />
            <xs:element type="xs:string" name="reference"
                />
            <xs:element type="MediaWikiType" name="
                MediaWiki" />
            <xs:element type="xs:string" name="added" />
        </xs:sequence>
        <xs:attribute type="xs:string" name="id"/>
        <xs:attribute type="xs:string" name="type"/>
    </xs:complexType>
    <xs:element name="key" type="KeyType"/>
</xs:schema>
```

The XSD structure for keys is shown in listing 5.12, where all the mentioned XML attributes and elements from listing 5.10 is also shown.
The XSD structure for key rings are missing at this point, but it is also needed to be defined.

Listing 5.13: Key Ring structure in XSD (XML Schema) format.

```
<xs:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified" xmlns:xs="http://www.w3
    .org/2001/XMLSchema">
<xs:complexType name="MediaWikiType">
    <xs:sequence>
        <xs:element type="xs:string" name="url"/>
        <xs:element type="xs:string" minOccurs="0"
            maxOccurs="1" name="username"/>
        <xs:element type="xs:string" minOccurs="0"
            maxOccurs="1" name="password"/>
        <xs:element type="xs:string" name="file-extension
            "/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="KeyType">
    <xs:sequence>
        <xs:element type="xs:string" name="symmetric-key"
            />
        <xs:element type="xs:string" name="IV"/>
        <xs:element type="xs:string" name="name"
            minOccurs="0" maxOccurs="1" />
        <xs:element type="xs:string" name="read-key"
            minOccurs="0" maxOccurs="1" />
        <xs:element type="xs:string" name="write-key"
            minOccurs="0" maxOccurs="1" />
        <xs:element type="xs:string" name="reference"/>
        <xs:element type="MediaWikiType" name="MediaWiki"
            />
        <xs:element type="xs:string" name="added" />
    </xs:sequence>
    <xs:attribute type="xs:string" name="id"/>
    <xs:attribute type="xs:string" name="type"/>
</xs:complexType>

<xs:element name="key-ring">
    <xs:complexType>
        <xs:sequence>
```

```
            <xs:element name="keys">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="key" type="
                            KeyType" minOccurs="0"
                            maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element type="xs:string" name="last-
                updated"/>
        </xs:sequence>
        <xs:attribute type="xs:string" name="id"/>
    </xs:complexType>
</xs:element>
</xs:schema>
```

The key ring XSD structure is shown in listing 5.13, and all the same elements and attributes as in both listing 5.11 and listing 5.10 is represented.
The methods used in regards to keys are not described in this section, as this section only include the structure of the keys. These methods will be described at a later section, where all methods will be described more in depth.

5.7 Key sharing

When sharing keys between users it is important that the key is not shared to an intruder and that the intruder can not read the them. The implementation of key sharing is done by using PGP. PGP is a hybrid crypto system combining both asymmetric cryptography as well as symmetric cryptography.
It starts by having two users: User A who wants to retrieve a file key from user B. User A starts by generating a key pair using asymmetric cryptography and sends the public-key to user B. Here it is assumed that the public-key shared between user A and user B has not been modified by any third part. User B generates a random key for encrypting the file key that is later sent to user A. User B encrypts the file key with the random generated key by using symmetric cryptography. User B also encrypts the random generated key by using the public-key retrieved from user A in the beginning. User B now sends the encrypted file key and encrypted random generated key, that was used for the encryption of the file key, to user A. User A can now, as the only one, decrypt the encrypted random generated key by using the private-key from the

key pair he generated in the beginning. When the random generated key has been decrypted it is possible to decrypt the file key with it and the key can be used by user A.
In the key sharing it is possible for the user to allow respectively read permissions and/or write permissions but only if the user have these permissions from the beginning.

5.8 Implementation description

The overview of the implementation has been described, the cryptography implementations as well as the structure of the keys. However, some methods have not been covered yet and this section is to somewhat cover the missing parts of the implementation. Such as methods for keys, the client and more.
Listings will be included to show the source code but only the more complex methods.

A console application has been implemented making sure it is easy for the user to encrypt, decrypt, verify, download, rename and upload files to the MediaWiki. Firstly looking at the initialization, that is shown in listing 5.14

Listing 5.14: Initialization of the system.

```
private void _init() {
    _localRootDir = new File(_config.getRootDirectory());
    File file = new File(_config.getLocalKeyPath());
    if (!file.exists() || !file.isFile()) {
        try {
            file.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    } else {
        this._key = Key.loadKeyFromXML(file);
    }
    if (this._key == null) {
        System.out.println("No key was found. Do you want
            us to create one for you (Y/N)? ");
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
        try {
            String create = br.readLine();
            if (create.equalsIgnoreCase("y")) {
                this._key = this._createKey(file);
            } else {
                file.delete();
                return;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    } else {
        System.out.println("Key␣was␣succesfully␣loaded.")
            ;
    }
    if (!_localRootDir.isDirectory()) {
        _localRootDir.mkdir();
        System.out.println("Couldn't␣find␣the␣defined␣
            root␣dir,␣but␣it␣has␣been␣created.");
    }
    file = this._key.download(this._localRootDir.getPath
        (), true);
    File newFile = new File(this._localRootDir.getPath()
        + "/" + this._key.getName() + ".xml");
    newFile = this._key.decrypt(file, newFile, true);
    if (newFile != null) {
        this._keyRing = KeyRing.loadXML(newFile.getPath()
            );
        newFile.delete();
        this._clientLoaded = true;
    }
}
```

Rootdir and the key file is located in the beginning, and if the key file does not exists the user is asked whether or not they want to create a new key. If the rootdir does not exist it is created as well.
java.io.Bufferedreader is used for getting inputs from the user.
A key is included in the configration file that the system reads from in the initialization. This key is linked to an encrypted key ring stored on the MediaWiki. This key ring is downloaded and decrypted.

Next thing is the uploadFile method shown in listing 5.15.

Listing 5.15: Upload file method.

```
public void uploadFile(File file, boolean moveFile) {
    if (!file.exists()) {
            return;
        }

        System.out.println("\nAttempting to upload file: " +
            file.getName());
        Key key = this._keyRing.getKeyByFilename(file.getName
            ());

        if (key == null) {
            key = new Key(file);
            this._keyRing.addKey(key);
            this._key.encryptAndUpload(this._keyRing);
            key.encryptAndUpload(file);
        } else {
            if (key.getWriteKey() == null || key.getWriteKey
                ().isEmpty()) {
                System.out.println("\tMissing permission. You
                    do not have write access to the current
                    file.\n\tFile can not be uploaded.");
                return;
            } else {
                if (key.encryptAndUpload(file)) {
                    System.out.println("\t" + file.getName()
                        + " was uploaded with existing key.");
                }
            }
        }
        if (moveFile) {
            try {
                InputStream in = new FileInputStream(file);
                OutputStream out = new FileOutputStream(this.
                    _localRootDir.getPath() + "/" + file.
                    getName());
                IOUtils.copy(in, out);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Looking at listing 5.15 it looks if there exists a key, if not a new key is created and added to the user's key ring, encrypting and uploading the file with this key. Otherwise the existing key is used for encrypting and uploading, as long

as there exists a non empty write-key.
The moveFile boolean input is used for whenever the user wants to upload a file outside of the rootdir. In the end it then copies the file to the rootdir, if this is true.

The next method is the renameFile, which is shown in listing 5.16.

Listing 5.16: Client systems rename file method.

```
public void renameFile(String oldName, String newName) {
    File oldFile = new File(_localRootDir.getPath() + "/"
        + oldName);
    File newFile = new File(_localRootDir.getPath() + "/"
        + newName);

    if (newFile.exists()) {
        System.out.println("\t" + newName + "␣already␣
            exists,␣delete␣or␣try␣another␣name.");
        return;
    }
    Key key = this._keyRing.getKeyByFilename(oldName);
    if (key == null) {
        System.out.println("Key␣could␣not␣be␣found");
    }
    if (oldFile.renameTo(newFile)) {
        System.out.println("\t" + oldName + "␣was␣
            succesfully␣renamed␣to␣" + newName);
        key.setName(newName);
        KeyRing keyRing = this._keyRing.
            getKeyRingContainingKey(key.getId());
        keyRing.addKey(key, true);
        if (this._key.getReference().equals(keyRing.getId
            ())) {
            key = this._key;
        } else {
            key = this._keyRing.getKeyReferingToKeyRing(
                keyRing.getId());
        }
        key.encryptAndUpload(keyRing);
    } else {
        System.out.println("File␣was␣not␣renamed.␣Try␣
            again.");
    }
}
```

Looking at listing 5.16 it can be seen that the key for the file is looked up. Afterwards the name of the key is changed to the new name, added to the key ring. The key that is referring to the key ring (earlier called parent key) is found and used to encrypt and upload the key ring.

The last method described is when a user wants to export a key to another user, meaning it is a part of the key sharing. This is done by either making an exact copy of the key (if it should contain all the same permissions) or just include for example a read-key. This new key is then encrypted using a random generated symmetric-key and initialization vector (IV). The symmetric-key and IV is then encrypted with the given public-key from the other user.
These are the most difficult methods from the Client.java explained. There are other methods in the same file as well, however these are very easy to understand and does not need further explanation.
The function to do this can be seen in listing 5.17.

Listing 5.17: Client exportKey method.

```
Key newKey = key.exportKey(read, write); // Exports the
    key including either read and/or write-key

Asymmetric asym = new Asymmetric(public-key);
String secret = Symmetric.generateKey();
String IV = Symmetric.generateiv();
Symmetric sym = new Symmetric(secret, IV);
File decryptedFile = new File(newKey.getId());
newKey.saveXML(decryptedFile);

InputStream in = new FileInputStream(decryptedFile);
OutputStream out = new FileOutputStream(file);
sym.encrypt(in, out);

decryptedFile.delete();

return asym.encryptByUsingpublic-key(secret + " : " + IV)
    ;
```

5.8.1 Configuration file

The configuration file is an XML document which the program will read as soon as it is started. The configuration allows the user to specify where the keys and files are going to be stored and the MediaWiki endpoint. The configuration contains following:

- Root directory: Directory containing all the files downloaded from the cloud.
- Local key: The key used to find, decrypt and encrypt the key ring stored in the cloud.
- MediaWiki: MediaWiki information such as url, file extension allowed and optional username and password.

That gives structure of the Config.xml shown in listing 5.18.

Listing 5.18: Example of Config structure in XML.

```
<config>
        <root-directory>testFUSE/</root-directory>
        <local-key>Localkey.xml</local-key>
        <MediaWiki>
                <url>https://localhost/MediaWiki</url>
                <username>user1</username>
                <password>user1</password>
                <file-extension>.box</file-extension>
        </MediaWiki>
</config>
```

An XSD (XML Schema) has also been added to prevent program failure if the user mistype or forgets some of the required XML elements.

Listing 5.19: Config structure in XSD (XML Schema) format.

```
<xs:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified" xmlns:xs="http://www.w3
    .org/2001/XMLSchema">
    <xs:complexType name="MediaWikiType">
        <xs:sequence>
            <xs:element type="xs:string" name="url"/>
            <xs:element type="xs:string" minOccurs="0"
                maxOccurs="1" name="username"/>
            <xs:element type="xs:string" minOccurs="0"
                maxOccurs="1" name="password"/>
            <xs:element type="xs:string" name="file-
                extension"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="config">
        <xs:complexType>
            <xs:sequence>
                <xs:element type="xs:string" name="root-
                    directory"/>
                <xs:element type="xs:string" name="local-
                    key"/>
                <xs:element type="MediaWikiType" name="
                    MediaWiki"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

5.8.2 Key methods

The structure of keys have been established in an earlier section of the implementation, but the methods have not been covered yet. This section will only cover the complex method and not the most basic ones.
When a user wants to upload a file, this is done through the key method uploadFile. This is shown in listing 5.20.

Listing 5.20: Key method uploadFile.

```
private boolean _uploadFile(File encryptedFile) {
    boolean success = false;
    try {
        Mediawiki wiki = this._mediaWiki.login();
        InputStream in = new FileInputStream(
            encryptedFile);
        String fileName = this._reference + this.
            _mediaWiki.getFileExtension();

        String encryptedFileHash = this.
            _generateEncryptedFileHash(in);
        if (encryptedFileHash == null ||
            encryptedFileHash.isEmpty()) {
             in.close();
             throw new IllegalStateException("Could not 
                 encrypt file hash");
        } else {
            String content = wiki.getPageContent("File:"
                + fileName);
            if (content == null || !content.equals(
                encryptedFileHash)) {
                in = new FileInputStream(encryptedFile);
                try {
                    wiki.upload(in, fileName,
                        encryptedFileHash, "");
                    wiki.edit("File:" + fileName,
                        encryptedFileHash, ""); // Have to
                          edit page because upload does not
                          change content but only adds.
                    success = true;
                } catch (Exception e) {
                    e.printStackTrace();
                    throw new IllegalStateException();
                }
                in.close();
            } else {
                System.out.println("File␣is␣no␣different␣
                    from␣the␣file␣uploaded␣earlier.␣Upload
                    ␣has␣been␣cancelled.");
            }

            encryptedFile.delete();
        }

        this._mediaWiki.logout();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return success;
}
```

The uploadFile is returning a boolean, whether or not the file has been successfully uploaded. First logging into the MediaWiki, using the information given from this key and getting the file extension defined. Secondly the file hash is generated using the input stream of the file and then encrypted using the write-key.
The page information from MediaWiki for this exact file is then found. This is used to find the exact location of uploading, and afterwards the page content is changed to include the encrypted file hash. This is done using the Medi-

aWiki command edit, instead of uploading. The encrypted file is deleted locally afterwards, and then logging out of the MediaWiki to delete the token used.

Before uploading to the MediaWiki it has to be encrypted and this is shown in the listing 5.21.

Listing 5.21: Key method encryptFile.

```
private File _encryptFile(File decryptedFile) {
    try {
        File encryptedFile = new File(this._reference +
            this._mediaWiki.getFileExtension());
        Symmetric sym = new Symmetric(this._symmetricKey,
             this._iv);
        InputStream fileKeyInput = new FileInputStream(
            decryptedFile);
        OutputStream fileKeyOutput = new FileOutputStream
            (encryptedFile);
        sym.encrypt(fileKeyInput, fileKeyOutput);
        fileKeyInput.close();
        fileKeyOutput.close();

        return encryptedFile;
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}
```

The method shown in listing 5.21 is getting a non encrypted file as input and returns the encrypted file. Finding the encrypted file's extension, from the key's MediaWiki information.
Simply opening the input stream from the non encrypted file, to the output stream of the encrypted file, and uses the two streams together with the Crypto.Symmetric to encrypt the stream. This will encrypt the output file, and this file is returned in the end.

At some point the user wants to download the uploaded files, this is done using the method download, shown in listing 5.22.

Listing 5.22: Key method download.

```
public File download(String path, boolean
    runThroughRevisions) {
```

```
    if (!path.endsWith("/")) {
        path += "/";
    }
    Mediawiki wiki = this._mediaWiki.login();
    String pageName = "File:" + this._reference + this.
        _mediaWiki.getFileExtension();
    String url = null;
    String content = null;
    try {
        url = wiki.getImageInfo(pageName).getUrl();
        content = wiki.getPageContent(pageName);
    } catch (Exception e) {
        e.printStackTrace();
        this._mediaWiki.logout();
    }
    File download = this._downloadFile(url, content, path
        );
    if (download == null) {
        if (runThroughRevisions) {
            try {
                download = this._findCorrectFile(wiki.
                    getRevisions(pageName), path);
            } catch (Exception e) {
                e.printStackTrace();
                this._mediaWiki.logout();
                throw new IllegalStateException("File␣
                    does␣not␣match␣file␣content");
            }
        } else {
            throw new IllegalStateException("File␣does␣
                not␣match␣file␣content");
        }
    }

    this._mediaWiki.logout();

    return download;
}
```

Looking at listing 5.22 it can be seen that first the application is logging into the MediaWiki, then the page information for the file is given. Here the reference-attribute from the key is used.
Using the getImageInfo to get the direct url to the file. When having the url a

private-key method _downloadFile is used, that basically opens an input- and output stream to read the file and write it locally (see section 5.3 to read more about file streams). The _downloadFile method might return null, if this version of the file could not be validated. If this happens, as it can be seen in listing 5.22 it will find the last valid revision. The last valid revision is found by looping through all revisions ordered by its upload date descending. In the end logging out of the MediaWiki to delete the token used and returning the newest revision that could be verified.

The last thing to show of the key methods is the decrypt and this is shown in the listing 5.23.

Listing 5.23: Key method decrypt

```
private File _downloadFile(String url, String content,
    String path) {
     File file = new File(path + this.getReference() +
         this._mediaWiki.getFileExtension());
     try {
          InputStream in = new URL(url).openStream();
          if (content == null || content.isEmpty() || !this
              ._validateFile(in, content)) {
               in.close();
               return null;
          } else {
               in = new URL(url).openStream();
               OutputStream out = new FileOutputStream(file)
                   ;
               IOUtils.copy(in, out);
               out.close();
          }
          in.close();

          return file;
     } catch (IOException e) {
          e.printStackTrace();
     }

     return null;
}
```

Looking at listing 5.23 it can be seen that the Crypto.Symmetric library is used and this is initialized by using the information from this key. Opening the two file streams input- and output stream to read the encrypted file and write

the file decrypted (see section 5.5.2 to read about the decryption process using symmetric cryptography). Deleting the encrypted file and returning the newly decrypted file.
This was the more complex methods of the key library. However, the library does include many more methods, but these are all simple and easy to understand.

5.8.3 FUSE

In this implementation it is meant to work such that the application on the computer to detect changes to files and automatically upload changes to the MediaWiki.
The current application is implemented in Oracle's Java, and the regular FUSE API, is used for C, however there does exist libraries to make it Java compatible. In this project, the library chosen for FUSE in java is called FUSE-JNA (see [Eti]).
There have been problems with the implementation of the FUSE file system and there for the application does not include the FUSE mounting part. However, the source is still there, for further working possibilities. The reason for the problem is explained in the end of this section.
When starting the application the folder should be mounted, and have downloaded all the files that the user have keys for. The mount method does whatever needed before mounting and this is shown in listing 5.24.

Listing 5.24: The FUSE library mount method.

```
public void mount() throws FUSEException {
    System.setProperty("jna.nosys", "true");
    String rootdirPath = _localRootDir.getPath();
    if (!_localRootDir.exists()) {
        _localRootDir.mkdir();
    }
    if (!_isDirEmpty(rootdirPath)) {
        System.out.println(rootdirPath + "␣is␣not␣empty.␣
            The␣root␣directory␣folder␣needs␣to␣be␣empty.\
            nConsider␣choosing␣a␣new␣rootdir␣by␣changing␣
            the␣Config.xml.\nMounting␣of␣" + rootdirPath +
             "␣Canceled.");
        return;
    }
    _loadFileKeys();
    this._addFilesDir(rootdirPath);
    this.mount(_localRootDir.getPath());
}
```

Looking at listing 5.24, the rootdir is checked at first and created if it does not exist. The actual rootdir have to be empty when mounting it, so this is checked. After this the keys are loaded, meaning they are downloaded from MediaWiki, and the corresponding files are downloaded as well.
After the files are downloaded, they are added to the rootdir, this is not copying them into the folder, but creating a MemoryFile object. This is how the FUSE-JNA works with files. This object is then added to the rootdir list of files. Then when mouting the rootdir, the files gets added to the mounted file system and the user can work with them.

When the file system is mounted, the user can work with files, and while doing that the system should automatically upload files. However, when uploading files through the FUSE library, it gives problems as explained earlier.
Looking back at the implementation overview from figure 5.1, it can be seen that there is use of the jaxb xml parser, and this is causing problems when working in FUSE as it gives parser errors. When uploading to the MediaWiki, XML is used to send information, and somehow the FUSE library cannot handle this way of doing it, as it gives ClassNotFoundException as seen in listing 5.25.

Listing 5.25: Error given when uploading files via FUSE.

```
javax.xml.bind.JAXBException
- with linked exception:
[java.lang.ClassNotFoundException: org/eclipse/
    persistence/jaxb/JAXBContextFactory]
```

Due to the error from listing 5.25, the FUSE part was removed from the running application, but the Java code is still in the project, as this problem has been close to resolved.

5.9 Summary

The implementation section has described how the implementation of the application has been done. First with an overview of how the whole setup have been done. Next how some different problems have been solved such as encoding, file streams and the unique identifier. The cryptography library implementation was described. The key structure was described showing the XML structure and next the key sharing as well.
Implementation description covered a more technical explanation of the different methods implemented in this project including different listings. However, this only included some of the more complex methods.

Chapter 6

Results

This section should include Evaluation as well.

6.1 Testing

Testing is an important aspect when creating software to make sure the functionality works every time a new feature is implemented. To make sure the core functionalities work there have been created unit tests. That gives the following automated tests:

- Hashing
- Encoding
- Asymmetric cryptography
- Symmetric cryptography
- Key (including connection to MediaWiki)
- KeyRing (including connection to MediaWiki)

All tests included in the source code have been ran each time a new feature has been changed or implemented to make sure it was not breaking anything.

The most important tests are to make sure the cryptographic methods work by testing the hashing, symmetric- and asymmetric cryptography. The tests written here makes sure following is correct:

- Generating file hash.
- Comparing file hashes.
- Generating and retrieving public- and private-key.
- Asymmetric encryption using public-key.
- Asymmetric decrypting using private-key.
- Asymmetric encryption using private-key.
- Asymmetric decrypting using public-key.
- Generating and retrieving symmetric-key and IV.
- Symmetric encrypting using symmetric-key and IV.
- Symmetric decrypting using symmetric-key and IV.
- Encoding data.*
- Decoding data.*

*Encoding and decoding data is not cryptographic methods, but is used to send and store data.

Now that the cryptographic methods are tested to work, they can be used in keys and key rings. This gives the following tests in keys and key rings:

- Initializing a key pointing to a key ring.
- Initializing a key pointing to a file.
- Encrypting the file/key ring pointed to by the key.
- Decrypting the file/key ring pointed to by the key.
- Upload file/key ring using the key.
- Downloading and decrypt file/key ring user the key.
- Retrieving a key from a key ring.
- Inserting a key into a key ring.
- Sharing a key with only read permission.
- Sharing a key with read and write permissions.

Most of these tests have been ran several times to measure performance for different operations (see appendix A.2). Performance was not focused in this project but looking at the performance tests they show a decent result overall.

Besides unit tests some manual tests have also been made (some of them are shown in appendix A.1):

- Initialize system without a local key.

- Initialize system with a local key.
- Create and upload a file.
- Change and upload a file.
- Rename a file.
- Download and decrypt all files stored in the MediaWiki.
- Closing the system and delete all files locally.
- Sharing a file.

6.2 Threats to the system

A risk analysis has been made to list the threats to the system looking at the probability for the threat to happen and the impact of the threat.

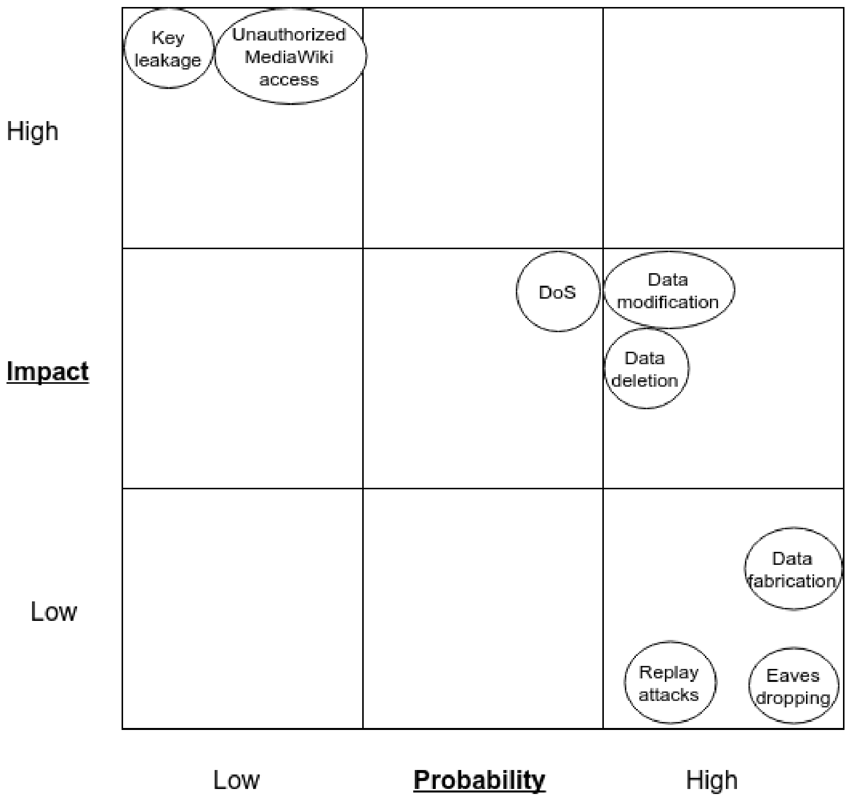


Figure 6.1: Risk analysis

6.2.1 Man-in-the-middle attacks

Man-in-the-middle attacks, also called MITM attacks, are attacks with most probability. A MITM is an attack where the attacker secretly modifies or alters the communication between two parties. The two parties have no idea that they are communicating through a MITM and think they are communicating directly with each other.
The reason the probability is that high for MITM attacks is that it is easy to perform a MITM attack on a unencrypted connection between two parties which in this exact case are the client and MediaWiki server.

6.2.1.1 Replay Attack

Replay attack is an attack where a MITM performs an earlier valid data transmission repeated or delayed.
In the case where a user uploads a file to the MediaWiki server and this data transmission is intercepted by a MITM. The MITM later repeats the data transmission to the MediaWiki server. This would upload the file successful to the MediaWiki server, but without any harm because the data is still valid. On the other hand, if it was the case that the MITM had intercepted an even older data transmission and repeated it to the MediaWiki server when a user wanted to upload a newer file it set the old file as the new file. Next time the user would download the file he would simply get an older file than the one he uploaded earlier. This could happen but thanks to revisions in MediaWiki the user would be able to retrieve his file in another revision.
This gives a low impact but a high probability to happen.

6.2.1.2 Eavesdropping

Eavesdropping is where a MITM intercepts the communication between two parties. This allows the MITM to simply listen to all the data communicated between two parties. In the case where a user uploads a file to the MediaWiki server it would allow the MITM to listen to the data sent to the MediaWiki server.This can easily happen but thanks to encryption done by the user before the data transmission, the MITM can not read the data. The same goes when a user downloads a file from the MediaWiki server.
This gives a low impact but a high probability to happen.

6.2.1.3 Deletion

A MITM has the option to delete the entire data transmission done via the communication between two parties. This allows the MITM to avoid any file uploads and/or file downloads to and from the MediaWiki server. This gives a bigger problem for the user because the user could lose data if he does not know the data uploaded earlier has been deleted. The MediaWiki server would never know that the user have tried to upload a file to it and can not create any revisions for the file.
This gives a medium impact but a high probability to happen.

6.2.1.4 Modification

A MITM have the option to modify the data transmitted via the communication between two parties. This allows the MITM to modify the file uploaded and/or file downloaded to and from the MediaWiki server. The biggest problem here is when the user tries to upload the file to the MediaWiki server because it could turn the valid file upload into an invalid file. When the user wants to download the file again the user would quickly find out that the data downloaded does not match with the checksum provided next to the file. This gives the same problem as deletion, in section 6.2.1.3, because the user could lose data if he does not know the file uploading had been modified.
This gives a medium impact but a high probability to happen.

6.2.1.5 Fabrication

A MITM can perform fabrication by transmitting data to the MediaWiki server. This allows the MITM to upload files to the MediaWiki server and upload files to the user when the user tries to download files.
If the user tries to download a file from the MediaWiki server where a MITM have been doing fabrication, the user would quickly find out that the data downloaded does not match with the checksum provided next to the file. The user then knows that this is a corrupt file and downloads another revision of the file.
This gives a low impact but a high probability to happen.

6.2.2 Denial-Of-Service attack

The Denial-Of-Service attack, also known as DoS attack, is when an attacker in some way deny the service needed for the user. It could be by giving the MediaWiki server so many requests, the server cannot handle and then causes it to break down or that the attacker simply uploads so many versions of an existing file, that it take a long time for the client to find a viable revision. Another type of DoS attack is a DDoS attack, Distributed-Denial-of-Service attack, where the principles are the same as a DoS attack but, where DoS attack typically uses one computer to do this with, DDoS attack uses multiple computers and are often global attacks distributed via a botnet.
A good thing about the MediaWiki server, is that it is made to handle a lot of traffic and this also makes it difficult to require a bot net big enough to take it down, however it is not impossible and it is getting easier and easier day by day to do. The impact is set to medium because taking the server down could only be temporarily one way or another and even so the user could still have their files locally stored, meaning it would only have impact until the server is up and running again.

6.2.3 Unauthorized MediaWiki access

Unauthorized MediaWiki access is when an attacker somehow get access to the server he or she is not supposed to have. This could either be by getting physical access to the server or by getting an sensitive user information (username and password) to the MediaWiki. When having administrator rights to a MediaWiki one is simply able to delete all files uploaded and this would be a disaster. However, the probability of this is very low if the password is strong, but then again this depends on the user administrating the MediaWiki. This could be improved by having a backup server available.
Getting physical access for a regular attacker is also very difficult and there for the probability for this happening is set to low on the risk analysis. The impact is high for this as well, because if an attacker got access to the server he or she could simply destroy it. This could like the administrator rights be improved by having a backup server available in a different building with different access requirements physically.

6.3 Future Work

This section is supposed to be a way to first of all show which parts of the project that was not implemented as intended. Next of all how they should have been and how any new contributors to this project can work further on.

6.3.1 FUSE & MediaWiki

Starting this future work section with the already mentioned FUSE & MediaWiki error that this implementation has. Instead of the user actively have to upload changes to the MediaWiki using the command line application, FUSE should be able to automatically detect changes to files and then upload the file. When doing this it gives an error which is caused by the XML binder jaxb.
Future work should include fixing this error, as it would give a complete file system and that gives a better user experience and it would look closer to already existing file cloud storage solutions such as DropBox.

6.3.2 Key Sharing made easy

The user experience as it is in the current application regarding key sharing, is not very user friendly. The two users have to manually do the most operations other than generating the needed keys, as well as sending the encrypted key file them self. Future work could be to automatically do most of these operations. However this would require some key distribution, through a trusted server to verify that a given public-key is in fact this users public-key. The MediaWiki servers could possibly be used to the key distribution, however this require some server development, and the goal of the project would be compromised in this way, as the clean implementation is avoided.
Besides that it is not known by the user who else has access to the key and what permissions they have.

6.3.3 Lost key

If a user looses their key they can not decrypt their files they have uploaded earlier, except if they shared all their files with another user. If this is the case the other user can simply share the key with the first user. If it is not the case that the user has shared the files with other users it is not possible for the user

to decrypt the files.
This could have been done by using secret sharing where the user divides the key into X number of junks and distributes each junk to a participant. This allows the user to reconstruct the key by combining Y number of shares together, where Y is less or equal to X, and the individual shares are of no use on their own.

6.4 Summary

The result section includes various tests that are included in the project to make sure the needed methods are rightfully implemented. Some of them was done as unit tests in the source code, and some of them where done by manually testing. The security aspect of the project was also looked at, and some of threats to the system was identified and they where included in a risk analysis.
In the end a future work section including various suggestions on how the project could be worked at in the future and how it could be made a better project.

Chapter 7

Conclusion

The goal of the project was to look into the possibility to develop a cloud based file storage environment using only a combination of cryptographic access control and key rings to do so. The idea was to use FUSE as the local client / file system and MediaWiki as the remote file storage server.

There have been implemented a solid prototype to show that it is possible to do so. The application does not need to trust the MediaWiki server as everything is encrypted client side, and it is not possible to use the current client to upload without the data get encrypted.
One of the goals was to use FUSE as the local client file system, but due to XML parser errors this goal was not reached. However, this could possibly be implemented in the future.
This also means that the current application should not be looked at as an end product, as it is a console application with the user telling the system which operations needed to be done.

The possibility to share files / keys in between users was a goal in this project as well and this goal have somewhat been reached. However, it is not an easy way to share files / keys, as this is done with a series of manual steps.

Furthermore, a risk analysis has been made and shows that the application is mostly compromised to DoS attacks.

Overall we are satisfied with the prototype, although it could have been nice to have the FUSE implementation ready as well.

Appendix A

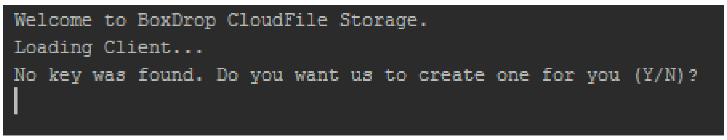
Appendix

A.1 Usage Guide

The purpose of the usage guide is to show how some of the basic operations are done in the current application, including screen shots.

A.1.1 Initialize system

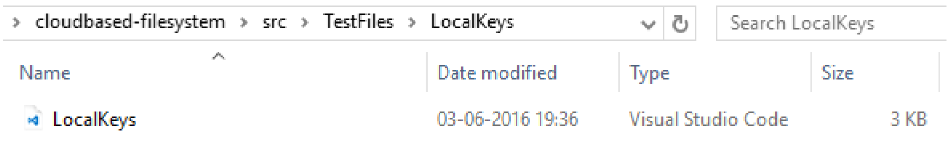
Assuming that MediaWiki have been setup and the needed information in the Config.xml file has been filled. The program needs to be started and because there is no key created for this user it will ask for it as shown on figure A.1.



```
Welcome to BoxDrop CloudFile Storage.
Loading Client...
No key was found. Do you want us to create one for you (Y/N)?
|
```

Figure A.1: Screenshot of when starting the system without a localkey.xml file.

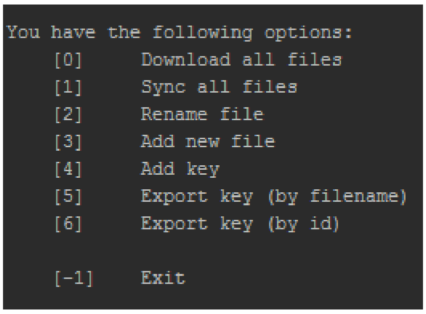
When inputting "y", keys file in at the given location from the config fileshown in figure A.2.



Name	Date modified	Type	Size
LocalKeys	03-06-2016 19:36	Visual Studio Code	3 KB

Figure A.2: Keys file created after typing y to the initialization.

After this the menu of the application is shown for the user, and this can be seen in figure A.3.



```
You have the following options:
    [0]     Download all files
    [1]     Sync all files
    [2]     Rename file
    [3]     Add new file
    [4]     Add key
    [5]     Export key (by filename)
    [6]     Export key (by id)

    [-1]    Exit
```

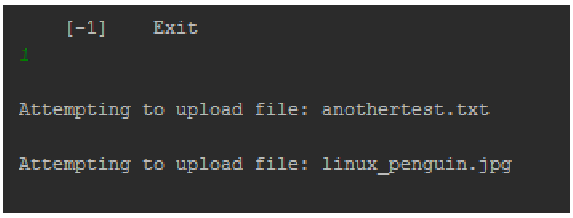
Figure A.3: Menu loaded after initialization and creation of the localkeys.xml

The keys file have been created for the user, and the menu is shown as in figure A.3 and this means that the system have been initialized successfully.

A.1.2 Create and upload file

When the system have been initialized and the menu is shown as in figure A.3, it is time for the user to create a file. This can either be done by using the "Add new file" feature inputting 3 or by putting a file into the root directory and use option 1 "Sync all files".

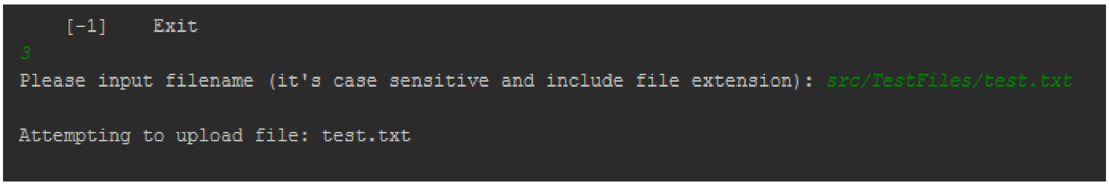
Firstly a file called linux_penguin.jpeg and anothertest.txt is added to the root directory and will be uploaded to the MediaWiki using the command "Sync all files". The output from the console application is shown in figure A.4.



```
    [-1]    Exit
1

Attempting to upload file: anothertest.txt

Attempting to upload file: linux_penguin.jpg
```

Figure A.4: The output from the application after synchronizing all files in the root directory.

Next a file that is not manually copied into the root directory by the user, is added. This is done using input 3 "Add new file" and the program want the file path to the file from where the java application is running. In this case it is src/TestFiles/test.txt. This is shown in figure A.5.



```
    [-1]    Exit
3
Please input filename (it's case sensitive and include file extension): src/TestFiles/test.txt

Attempting to upload file: test.txt
```

Figure A.5: The input and the output from the application when adding a file placed outside the root directory.

It is also added into the root directory, and this is shown in figure A.6.

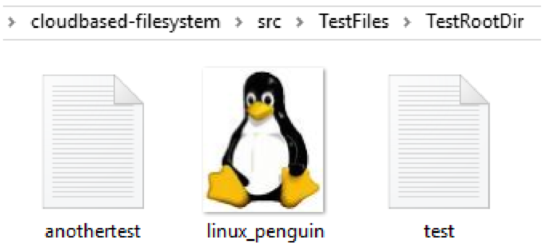


Figure A.6: The root directory after adding a file placed outside the root directory.

This is how to add one or more files after initialization of the system. Closing and opening the program will later be shown, to show that these files are in fact saved onto the MediaWiki.

A.1.3 Change and upload file

Three files are added to the root directory as shown in figure A.6, and now the content will be changed in anothertest.txt, as shown in figure A.7.

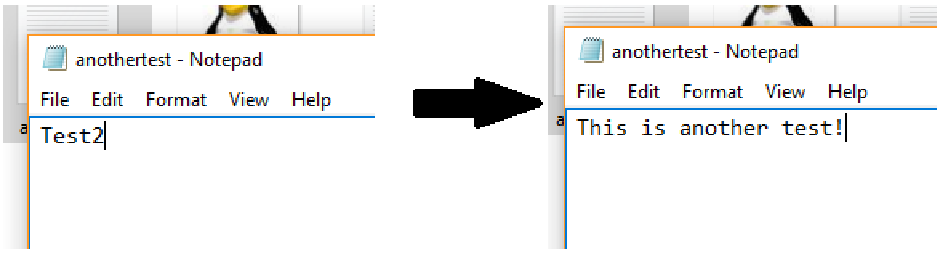
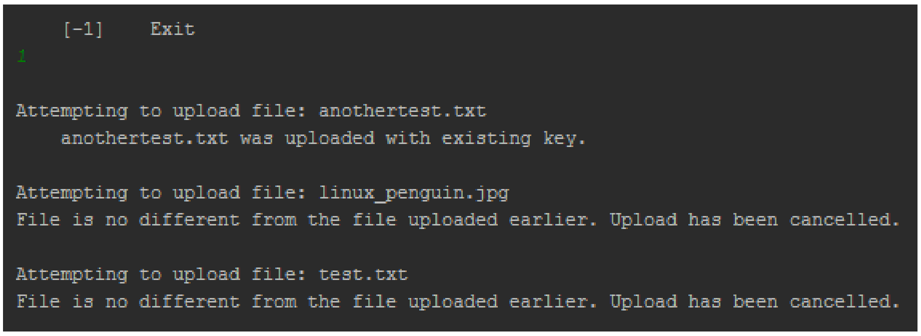


Figure A.7: Showing the change of anothertest.txt.

And afterwards the "Sync all files" needs to be inputted by the user. This is shown in figure A.8, where it can be seen that it only uploads anothertest.txt as this is the only one that have been changed.



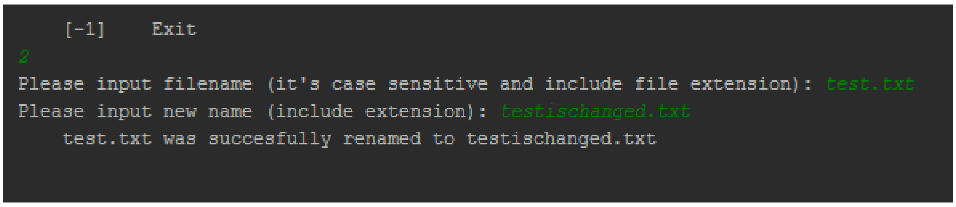
```
    [-1]    Exit
1

Attempting to upload file: anothertest.txt
    anothertest.txt was uploaded with existing key.

Attempting to upload file: linux_penguin.jpg
File is no different from the file uploaded earlier. Upload has been cancelled.

Attempting to upload file: test.txt
File is no different from the file uploaded earlier. Upload has been cancelled.
```

Figure A.8: Running the Sync all files command after having changed anothertest.txt

This was how to change a file in the current application, and later it will be shown that the changes actually have taken effect and are uploaded to the MediaWiki.

A.1.4 Rename file

Looking at the menu from figure A.3 it can also be seen that by inputting 2, it is possible to rename a file. This should be done by using this command, and not by doing it directly to the file. The user needs to input the name of the current file, and the new name that the user wants. Both input and output is shown in figure A.9.



```
    [-1]    Exit
2
Please input filename (it's case sensitive and include file extension): test.txt
Please input new name (include extension): testischanged.txt
    test.txt was succesfully renamed to testischanged.txt
```

Figure A.9: The input and output to and from the application when renaming a file.

The file have also been renamed in the root directory. This is shown in figure A.10.

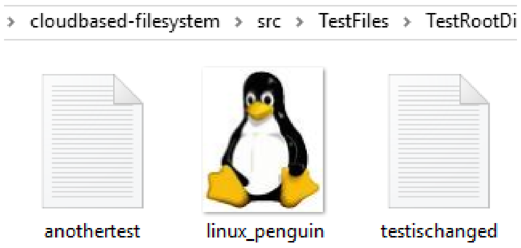
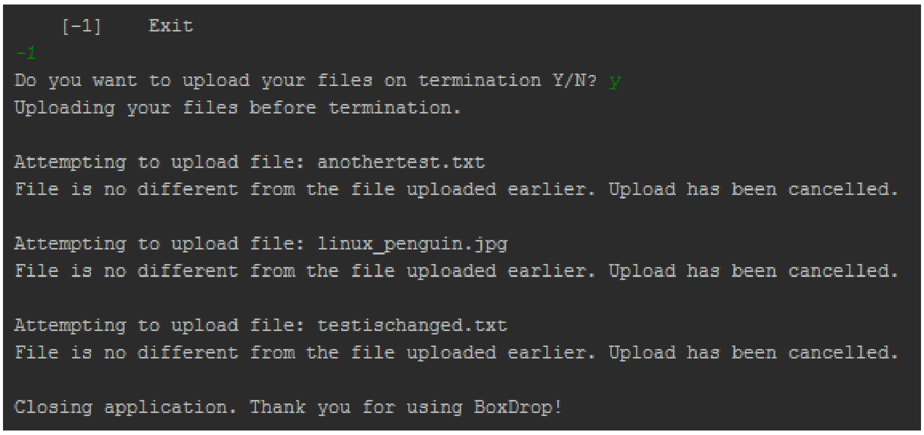


Figure A.10: The root directory after renaming test.txt to testischanged.txt .

This was how to rename a file the correct way when using the application. It will be shown that all these changes and rename will take effect when closing and starting the program.

A.1.5 Exit and start program

Now the session needs to be ended, which is done by inputting -1 into the application. After wards it will be shown that all the files are the same when opening the application again. Closing the session is shown in figure A.11.



```
    [-1]    Exit
-1
Do you want to upload your files on termination Y/N? y
Uploading your files before termination.

Attempting to upload file: anothertest.txt
File is no different from the file uploaded earlier. Upload has been cancelled.

Attempting to upload file: linux_penguin.jpg
File is no different from the file uploaded earlier. Upload has been cancelled.

Attempting to upload file: testischanged.txt
File is no different from the file uploaded earlier. Upload has been cancelled.

Closing application. Thank you for using BoxDrop!
```

Figure A.11: The input and output when ending the session.

It can be seen from figure A.11 that all files was already uploaded to the MediaWiki, but it attempted to do so anyway. Everything in the root directory have also been locally deleted as shown in figure A.12.

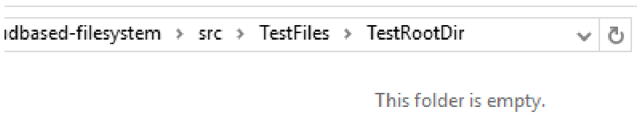
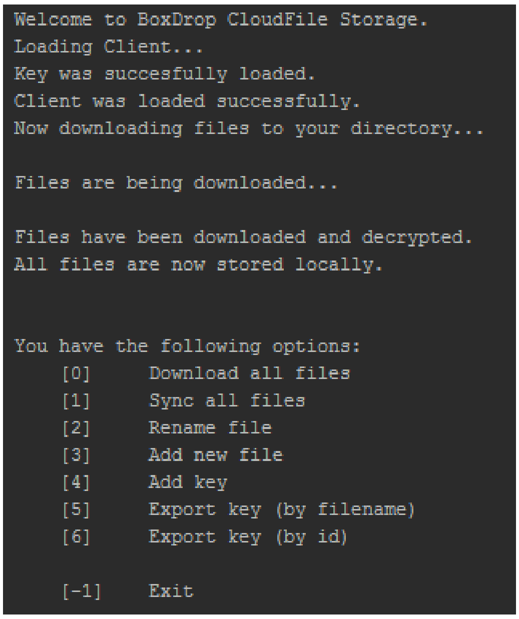


Figure A.12: The root directory is empty after closing the application and uploading all files.

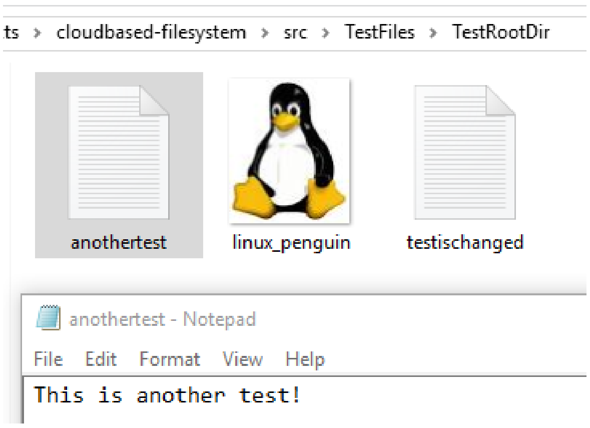
The application have been closed successfully, but to show that it actually works as intended the program needs to be started again. The start of the application again is shown in figure A.13.



```
Welcome to BoxDrop CloudFile Storage.
Loading Client...
Key was succesfully loaded.
Client was loaded successfully.
Now downloading files to your directory...

Files are being downloaded...

Files have been downloaded and decrypted.
All files are now stored locally.


You have the following options:
    [0]     Download all files
    [1]     Sync all files
    [2]     Rename file
    [3]     Add new file
    [4]     Add key
    [5]     Export key (by filename)
    [6]     Export key (by id)

    [-1]    Exit
```

Figure A.13: The output from the application when starting it, already having a localkeys.xml file.

All files is downloaded to the root directory and this is shown in figure A.14.


ts › cloudbased-filesystem › src › TestFiles › TestRootDir
anothertest
linux_penguin
testischanged
anothertest - Notepad
File Edit Format View Help

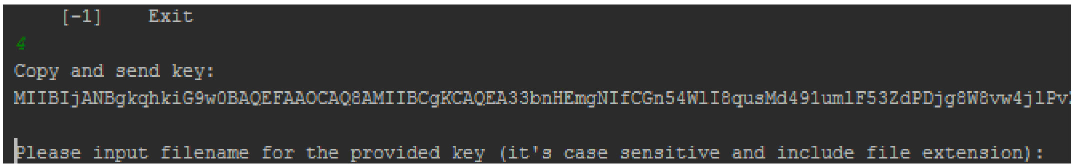
```
This is another test!
```

Figure A.14: The root directory shown after starting the application already having a localkeys.xml where some files has been uploaded to the MediaWiki. This including the change to anothertest.txt

Looking at figure A.14 it can also be seen that both the rename made in an earlier section as well as the change to anothertest.txt is the same.

A.1.6 Sharing file

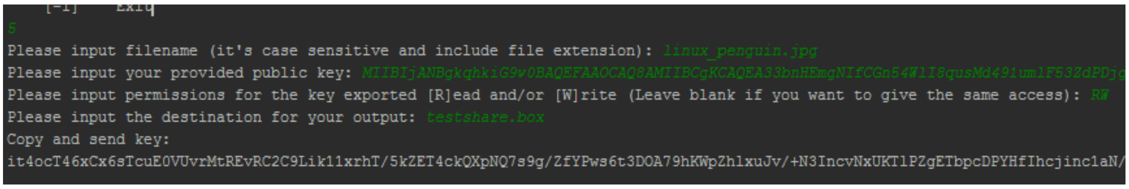
When sharing a file, it is needed for two users to have initialized the system. Firstly the user B use the Add Key command by inputting 4, this is shown in figure A.15.



```
    [-1]    Exit
4
Copy and send key:
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA33bnHEmgNIfCGn54W1I8qusMd491um1F53ZdPDjg8W8vw4j1Pv

Please input filename for the provided key (it's case sensitive and include file extension):
```

Figure A.15: User B typed Add Key, getting the secret that needs to be send to user A.

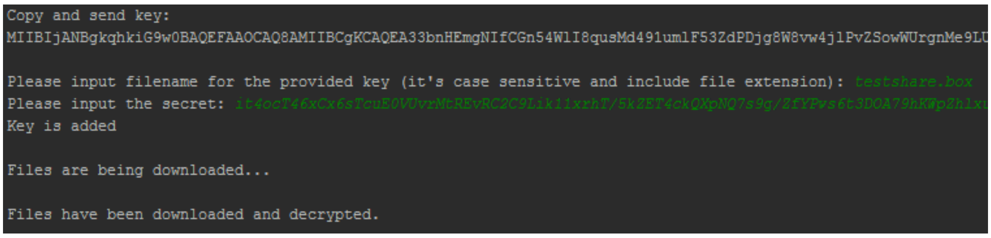
What user B needs to do now is to copy the public-key shown in figure A.15, and send it to user A over a secure line.
User A now uses the command "Export key (by filename)" with the linux_penguin.jpg file name as input and inputs user B's public-key received, as second input and "RW" as third to give read- & write access. This could also just be "R" for read access only. The last input is the destination for the output file.
When this is done the application outputs a testshare.box file and a one time generated secret that user B needs. User A'a input and output is shown in figure A.16.



```
    [-1]    Exit
5
Please input filename (it's case sensitive and include file extension): linux_penguin.jpg
Please input your provided public key: MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA33bnHEmgNIfCGn54W1I8qusMd491um1F53ZdPDjg
Please input permissions for the key exported [R]ead and/or [W]rite (Leave blank if you want to give the same access): RW
Please input the destination for your output: testshare.box
Copy and send key:
it4ocT46xCx6sTcuE0VUvrMtREvRC2C9Lik11xrhT/5kZET4ckQXpNQ7s9g/ZfYPws6t3DOA79hKWpZhlxuJv/+N3IncvNxUKTlPZgETbpcDPYHfIhcjinc1aN/
```

Figure A.16: User A exporting a key that needs to be send to user B along with the public-key of user A.

When user B inputs the given secret the file is downloaded as shown in figure A.17.



```
Copy and send key:
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA33bnHEmgNIfCGn54W1I8qusMd491umlF53ZdPDjg8W8vw4jlPvZSowWUrgnMe9LU

Please input filename for the provided key (it's case sensitive and include file extension): testshare.box
Please input the secret: it4ocT46xCx6sTcuE0VUvrMtREvRC2C9Lik11xrhT/5kZET4ckQXpNQ7s9g/ZfYPvs6t3DOA79hKWpZhlxu
Key is added

Files are being downloaded...

Files have been downloaded and decrypted.
```

Figure A.17: Application downloading the file given from user A to B.

Looking at the root directory of user B as shown in figure A.18 it can be seen that the linux_penguin.jpg picture have been downloaded successfully.

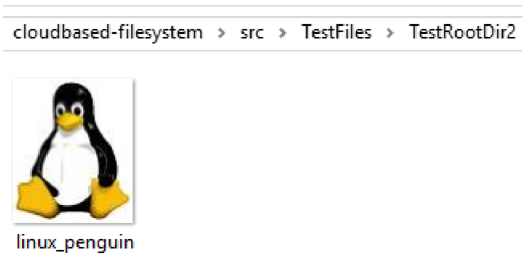


Figure A.18: The root directory for user B after getting a key from user A.

That is how the sharing of a file works in the current application.

A.2 Performance test

The purpose of the performance tests are to check that the program is not too slow. It is not to compare anything, but only to see that the program does run within regular operation times.

The tests have been ran on a machine running Ubuntu 16.04 with following specifications:

- Intel Core i5-5250U CPU @ 1.60GHz
- 16 GB memory
- Intel 120 GB SSD (SSDSCKHW120A4)

All tests are using a 104.9 MB test file and have been ran several times to find the average time of each operation.

A.2.1 Encryption using symmetric cryptography

This test includes reading a decrypted file and writing to a new file encrypting the content in the file:

Encrypting: $\sim 380ms$

Normal reading and writing without encryption: $\sim 120ms$

A.2.2 Decryption using symmetric cryptography

This test includes reading an encrypted file and writing to a new file decrypting the content in the file:

Decrypting: $\sim 600ms$

Normal reading and writing without decryption: $\sim 120ms$

A.2.3 Encryption using asymmetric cryptography

These tests include reading a hash and encrypting it by using public- and private key:

Encrypting using public key: $\sim 1ms$

Encrypting using private key: $\sim 10ms$

A.2.4 Decryption using asymmetric cryptography

These tests include reading an encrypted hash and decrypting it by using public- and private key:

Decrypting using public key: $\sim 1ms$

Decrypting using private key: $\sim 10ms$

A.2.5 Hash file

This test includes reading a file and generating a hash for the file:

Hashing: $\sim 800ms$

Reading without hashing: $\sim 75ms$

A.2.6 Encoding

This test includes reading a public- key and encode it:

Encoding: $\sim 1ms$

A.2.7 Decoding

This test includes reading a public- key encoded and then decode it:

Decoding: $\sim 1ms$

A.2.8 Upload file

This test includes reading a file locally and then upload it to a MediaWiki hosted locally:

Upload file without encryption: $\sim 5800ms$

Encrypt file and upload: $\sim 6500ms$

A.2.9 Download file

This test includes downloading an encrypted file from MediaWiki hosted locally:

Download file without decryption: $\sim 1750ms$

Download file and decrypt: $\sim 2400ms$

Bibliography

[Bla93] Matt Blaze. A cryptographic file system for unix. *Conference on Computer and Communications Security*, pages 9–16, 1993.

[CP97] G Cattaneo and G Persiano. Design and implementation of a transparent cryptographic file system for unix. 1997.

[dja] Django file storage. https://docs.djangoproject.com/en/1.9/howto/custom-file-storage/.

[EK] A. A. Elliott and G. S. Knight. Role explosion: Acknowledging the problem. http://knight.segfaults.net/papers/20100502%20-%20Aaron%20Elliott%20-%20WOLRDCOMP%202010%20Paper.pdf.

[Eti] EtiennePerot. Fuse-jna library. https://github.com/EtiennePerot/fuse-jna.

[Fou] Wikimedia Foundation. Wikipedia. https://www.wikipedia.org/.

[git] Git. https://git-scm.com/.

[Hal] Mike Halcrow. ecryptfs: a stacked cryptographic filesystem. http://www.linuxjournal.com/article/9400.

[HJ03] Anthony Harrington and Christian Jensen. Cryptographic access control in a distributed file system. *Proceedings of ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, pages 158–165, 2003.

[Hoh] Christoph Hohmann. Cryptofs. http://reboot.github.io/cryptofs/.

[Jor] Carole S. Jordan. A guide to understanding discretionary access control in trusted systems. http://fas.org/irp/nsa/rainbow/tg003.htm.

[Mag] Jeffrey B. Layton Linux Magazine. User space file systems. http://www.linux-mag.com/id/7814/.

[Mau] Ermelindo Mauriello. Tcfs: Transparent cryptographic file system. http://www.linuxjournal.com/article/2174.

[Meda] Mediawiki api imageinfo. https://www.mediawiki.org/wiki/API:Imageinfo.

[Medb] Mediawiki api revisions. https://www.mediawiki.org/wiki/API:Revisions.

[Medc] Mediawiki api upload. https://www.mediawiki.org/wiki/API:Upload.

[Medd] Mediawiki main page. https://www.mediawiki.org/wiki/API:Main_page.

[mys] Mysql. https://www.mysql.com/.

[nisa] Nist's policy on hash functions. http://csrc.nist.gov/groups/ST/hash/policy.html.

[NISb] NIST. Announcing the advanced encryption standard (aes). http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[NISc] NIST. Twirl and rsa key size. http://www.emc.com/emc-plus/rsa-labs/historical/twirl-and-rsa-key-size.htm.

[ofs] Ofs file storage. https://pythonhosted.org/ofs/.

[Ora] Oracle. Java. https://www.java.com/en/.

[php] Php. https://secure.php.net/.

[RA83] Shamir A Rivest, R. L and L Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 26(1):96–99, 1983.

[rem] Remotestorage.io. https://remotestorage.io/.

[Rog] Philip Rogaway. The security of desx. http://web.cs.ucdavis.edu/~rogaway/papers/cryptobytes.pdf.

[SS] IBM Sumit Singh, Software Engineer. Develop your own filesystem with fuse. http://www.ibm.com/developerworks/linux/library/l-fuse/.

[VCHS] Rick Kuhn Adam Schnitzer Kenneth Sandlin Robert Miller Vincent C. Hu, David Ferraiolo and Karen Scarfone. Guide to attribute based access control (abac) definition and considerations. `http://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.sp.800-162.pdf`.

[wik] Wikimedia foundation. `https://wikimediafoundation.org/wiki/Home`.

[Wri] Howard Wright. The encrypting file system – how secure is it? `https://www.sans.org/reading-room/whitepapers/win2k/encrypting-file-system-secure-it-211`.